NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

UNDERWATER ACOUSTIC NETWORKS: EVALUATION OF THE IMPACT OF MEDIA ACCESS CONTROL ON LATENCY, IN A DELAY CONSTRAINED NETWORK

by

José Manuel dos Santos Coelho

March 2005

Thesis Advisor:	Su Wen
Thesis Co-Advisor:	Geoffrey Xie
Second Reader:	John Gibson

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** March 2005	**3. REPORT TYPE AND DATES COVERED** Master's Thesis	
4. TITLE AND SUBTITLE: Underwater Acoustic Networks: Evaluation of the Impact of Media Access Control on Latency, in a Delay Constrained Network		**5. FUNDING NUMBERS**	
6. AUTHOR(S) **José Manuel dos Santos Coelho**			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000		**8. PERFORMING ORGANIZATION REPORT NUMBER**	
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A		**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**	
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited		**12b. DISTRIBUTION CODE**	

13. ABSTRACT (maximum 200 words)

This thesis presents an evaluation of the performance, in terms of throughput and latency, of two Media Access Control (MAC) mechanisms in Underwater Acoustic Networks (UANs), using a model designed in the COTS simulation tool OPNET 10.5. The carrier sense multiple access with collision avoidance is the predominant approach for implementing the MAC mechanism in UANs. However, the underwater acoustic environment is characterized by extreme propagation delays and limited bandwidth, which suggests that an Aloha-like scheme may merit consideration. The performance of these two schemes was compared with respect to two topologies: tree and grid. The results showed that an Aloha-like scheme that does not segment messages outperforms the contention-based scheme under all load conditions, in terms of both throughput and latency, for the two topologies. This thesis is the first to establish that Aloha-like MAC mechanisms can be more than a limited alternative for lightly loaded networks; more specifically, they can be the preferred choice for an environment with large propagation delays.

14. SUBJECT TERMS Underwater Acoustic Networks, Media Access Control, Collision Avoidance, Aloha, Network Simulation, Network Latency, Link Layer, Delay Constrained Networks			**15. NUMBER OF PAGES** 187
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

UNDERWATER ACOUSTIC NETWORKS: EVALUATION OF THE IMPACT OF MEDIA ACCESS CONTROL ON LATENCY, IN A DELAY CONSTRAINED NETWORK

José Manuel dos Santos Coelho
Lieutenant-Commander, Portuguese Navy
B.S., Naval Academy, Lisbon, 1990

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March 2005

Author: José Manuel dos Santos Coelho

Approved by: Su Wen
Thesis Advisor

Geoffrey Xie
Thesis Co-Advisor

John Gibson
Second Reader

Peter Denning
Chairman, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

This thesis presents an evaluation of the performance, in terms of throughput and latency, of two Media Access Control (MAC) mechanisms in Underwater Acoustic Networks (UANs), using a model designed in the COTS simulation tool OPNET 10.5. The carrier sense multiple access with collision avoidance is the predominant approach for implementing the MAC mechanism in UANs. However, the underwater acoustic environment is characterized by extreme propagation delays and limited bandwidth, which suggests that an Aloha-like scheme may merit consideration. The performance of these two schemes was compared with respect to two topologies: tree and grid. The results showed that an Aloha-like scheme that does not segment messages outperforms the contention-based scheme under all load conditions, in terms of both throughput and latency, for the two topologies. This thesis is the first to establish that Aloha-like MAC mechanisms can be more than a limited alternative for lightly loaded networks; more specifically, they can be the preferred choice for an environment with large propagation delays.

This thesis presents an evaluation of the performance, in terms of throughput and latency, of two Media Access Control (MAC) mechanisms in Underwater Acoustic Networks (UANs), using a model designed in the COTS simulation tool OPNET 10.5. The carrier sense multiple access with collision avoidance is the predominant approach for implementing the MAC mechanism in UANs. However, the underwater acoustic environment is characterized by extreme propagation delays and limited bandwidth, which suggests that an Aloha-like scheme may merit consideration. The performance of these two schemes was compared with respect to two topologies: tree and grid. The results showed that an Aloha-like scheme that does not segment messages outperforms the contention-based scheme under all load conditions, in terms of both throughput and latency, for the two topologies. This thesis is the first to establish that Aloha-like MAC mechanisms can be more than a limited alternative for lightly loaded networks; more specifically, they can be the preferred choice for an environment with large propagation delays.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

I would like to dedicate this thesis work to my wife, Maria, for her support and encouragement. Without her balance and motivation, this journey would be much more difficult. My dedication is extensive to my children, Carolina and Tomás. I am constantly amazed by their ingenious capacity for claiming for my attention, even in those moments where some work study was exhausting it.

I would like to thank my thesis advisor, Prof. Su Wen, the professor at NPS with whom I had more class hours, for her effort, support, and encouragement. My gratitude is also extended to my thesis co-advisor Prof. Geoffrey Xie, for constantly challenging my thought process, and to my second reader Prof. John Gibson, for his effort and mentorship.

I would like to thank the Portuguese Navy for giving me the opportunity for this unforgettable learning experience, and to NPS, professors, students, and staff, for providing it in a such welcoming environment.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. DEFINITION

An underwater acoustic network (UAN) consists of static and/or mobile nodes in a marine environment where the nodes communicate with each other using wireless acoustic channels. Typically, a special purpose node, called a "gateway," is equipped with an acoustic modem to communicate with the acoustic nodes, and with a high-speed data link to connect the UAN to command centers, hosted in ships or other types of installations. There is no assumption made regarding the nature of this high-speed link and, therefore, a command center or a data fusion center could be located anywhere else. A UAN may have more than one gateway. However, a typical implementation encompasses several sensor nodes that collect and send their data to relay nodes, which, in turn, forward that data in a hop-by-hop fashion to the gateway, whose sole function is to forward the data over a high-speed link (e.g., satellite, wireless, or even wire or fiber) [Xie 2001].

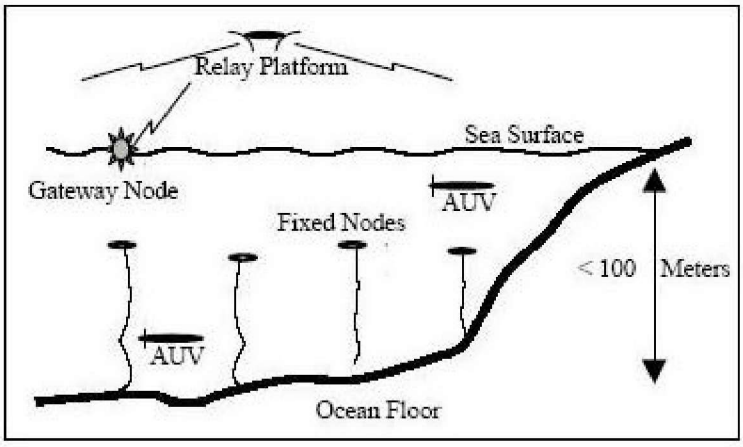

Figure 1. Nominal Underwater Acoustic Network [Xie 2001]

These networks have many similarities with mobile, ad hoc networks (often referred to by the acronym, "MANET"), in that they do not rely on some implemented infrastructure. Additionally, they may need to support node mobility, and rely on self-powered nodes. However, UANs have additional characteristics related with the idiosyncrasies of sound propagation in seawater: namely the propagation speed is altered

to have a nominal value of 1,500 meters per second (m/s), and the bandwidth available is extremely low (between 100 and 2,500 bits per second (bps), but typically 1,000 bps [Xie 2001]).

UANs are considered Delay Tolerant Networks (DTN), due to their extreme propagation delay. This class of networks is characterized by severe propagation delays, due to either link discontinuities or extreme propagations delays. UANs qualify for this classification, not because they suffer from systematic link discontinuities, whether planned or not, but because they are constrained by an extreme propagation delay [Gibson 2005b].

B. APPLICATION OF UNDERWATER ACCOUSTIC NETWORKS

There has been an increased interest in UANs, as evidenced by the recent growth of research projects and deployments of such networks. Examples include, the Deployable Autonomous Distributed System (DADS) and continental shelf observatories, such as the Front-Resolving Oceanographic Network with Telemetry (FRONT). The DADS network can support the participation of mobile nodes, such as manned submarines and Unmanned Undersea Vehicles (UUVs). It supports expeditionary operations in contested waters, with communication gateways to command centers that could be submerged, afloat, ashore, or possibly even those that are far away. Specifically, it can be used to provide inter-operability with anti-submarine warfare (ASW), intelligence surveillance reconnaissance (ISR), and meteorological oceanographic (METOC) systems [Rice 2002].

The tragic tsunami that occurred on Dec 26, 2004 in South Asia brought up another possible application of UANs. A tsunami wave typically has a very long wavelength, and an imperceptible height in the open seas, making it difficult to detect at sea level. Thus, an early warning system may be enhanced with the inclusion of underwater sensors that could detect the progression of such a wave and then forward the gathered data to a floating gateway equipped with a high-speed data connection (possibly a satellite link) to a command center.

Whether in a command and control environment, or in a simple data collection mode, UANs are required to perform efficiently in order to conserve the battery power while providing good network performance, measured by total throughput and average message latency.

C. PROBLEM STATEMENT

UANs present several unique challenges and problems. One of these problems is the media access control (MAC) mechanism in an aquatic acoustic wireless environment. The typical approach is to mimic the wireless aerial radio-based solution, using the contention-based control mechanism with the exchange of two control messages in order to reserve the medium. This approach has a significant impact on a network's performance, mainly due to the propagation delay associated with exchanging the two control messages. Whereas the penalty for such an exchange might be negligible in certain environments, e.g., in an aerial environment, it frequently has a more significant impact on network performance for underwater acoustic communication. When we compare the propagation speed in wireless aerial radio-based networks, $\approx 3 \times 10^8$ m/s – the speed of light, with the typical propagation speed in UANs, $\approx 1.5\ 10^3$ m/s – the speed of sound in seawater, the difference between them is five-orders of magnitude. This provides evidence that a more educated choice of the MAC mechanism is recommended in order to maximize the network performance.

A different approach based on *a priori* allocation of channels, which allows full-duplex communication between the nodes, was proposed in [Xie 2001], [Gibson 2002], and [Xie 2004]. The bandwidth is divided into an appropriate number of channels to allow full-duplex communication between the sensors and relay nodes, and between relay nodes. Some variations on this setting are presented in [Gibson 2005b], with a contention-based mechanism between the sensor and relay nodes, and contention-free full-duplex communication between the relay nodes.

When comparing the full-duplex mode with the contention-based approach, the full-duplex mode does not have the propagation delay penalty induced by the exchange of the control messages to reserve the channel. However, it does incur a transmission delay penalty due to the division of the available bandwidth into individual channels. Additionally, taking into consideration the characteristics of the network and the typical

network topology, a simple uncoordinated MAC mechanism may serve well the intended traffic pattern [Gibson 2005b]. Therefore, we need to understand better how the message latency in UANs having different topologies and serving different traffic patterns is affected by the chosen MAC mechanism.

D. OBJECTIVE AND SCOPE

The objective of this thesis is to evaluate and compare the performance of UANs with two different MAC mechanisms: namely the commonly used contention-based with collision avoidance and the uncoordinated contention-free mechanism, both of which are half-duplex, in terms of message latency. A complementary comparison is made in [Gibson 2005b] between the contention-based and the *a priori* allocated channels MAC mechanisms. However, the analysis there is based on a model with a very simple linear topology that, for example, does not allow traffic generation in intermediate nodes. This thesis addresses some of the limitations and constraints used by that model, with the creation of a more realistic model using the simulation tool, OpNet.

In essence, the thesis endeavors to answer the following questions:

- Considering a typical traffic load in UANs, which MAC mechanism being considered here renders less average end-to-end message delays?
- In a command and control environment (or other type of settings where we need to assure message latency predictability), how do the two media access mechanisms compare? Which one is more suitable for such an environment?

E. THESIS OUTLINE

This thesis begins with a background review of the constraints and limitations that affect underwater acoustic communications, the MAC mechanisms that have been proposed for UANs, and the related work comparing them. Chapter III describes the link-layer protocols modeled, the design choices, and the OpNet implementation. Chapter IV provides a description of the simulation design. Chapter V describes how we validated the model. Chapter VI and VII presents the simulation results for the tree and grid topologies, respectively. Chapter VIII discusses the results and presents some pertinent conclusions and recommendations for follow-up work to better understand the effect of the chosen MAC mechanism in message latency.

II. BACKGROUND AND RELATED WORK

A. MEDIA ACCESS LIMITATIONS, CONSTRAINTS, AND CHALLENGES

Although acoustic communications have been proven to be a viable way to provide underwater networking, there are some challenges related with their delay characteristics. These require the quest for alternative approaches regarding the MAC mechanism. Two of their key limiting factors are their severe propagation delays associated with the nominal 1500 m/s propagation speed of sound in seawater, and their constrained bandwidth, due to extreme attenuation of frequencies above 50 Kilohertz over any appreciable distance. Additionally, the omni-directional nature of the medium leads to the same hidden terminal and near-far problems associated with wireless aerial radio-based networks [Gibson 2005a].

1. The Current Media Access Control Approach in UANs

The most common approach to control the media access in UANs is akin to the wireless aerial network protocol, IEEE 802.11, the so-called Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA). The CSMA/CA method and the "stop-and-wait" flow control, with Automatic Repeat Request (ARQ) error recovery, typically implemented in UANs, severely limit the throughput and increase the network's latency. The CSMA/CA method implies the exchange of two small control messages, Request-to-Send (RTS) and Clear-to-Send (CTS), to reserve the medium, announcing to all the reachable neighbors the need for deferring their own communications in order to avoid the potential for collisions (i.e., overlapping data receptions). Although in a radio network environment the overhead of that exchange is small because the propagation delay is negligible, that is not the case with UANs. It has been shown that the impact of such exchanges in a simple backbone network can significantly increase the delay in message delivery [Xie 2001] [Gibson 2005b].

Figure 2 eloquently depicts the propagation delay penalty that handshake-type protocols incur. When Node A wants to forward data to Node C it needs to relay the data through Node B. Before sending the data, the sending and receiving nodes need to exchange the RTS-CTS control messages. Node B only initiates its control exchange with Node C after it receives the data from Node A. Thus, for each hop, two propagation delay

terms are added to the total delay for each data frame. Therefore, a lower bound for the total delay is a function of three times the propagation delay between nodes. In an environment where the main contribution for the total delay is precisely the propagation delay, this represents a strong downside. It is important to mention that in this elaboration we are not considering transmission or queuing delays.

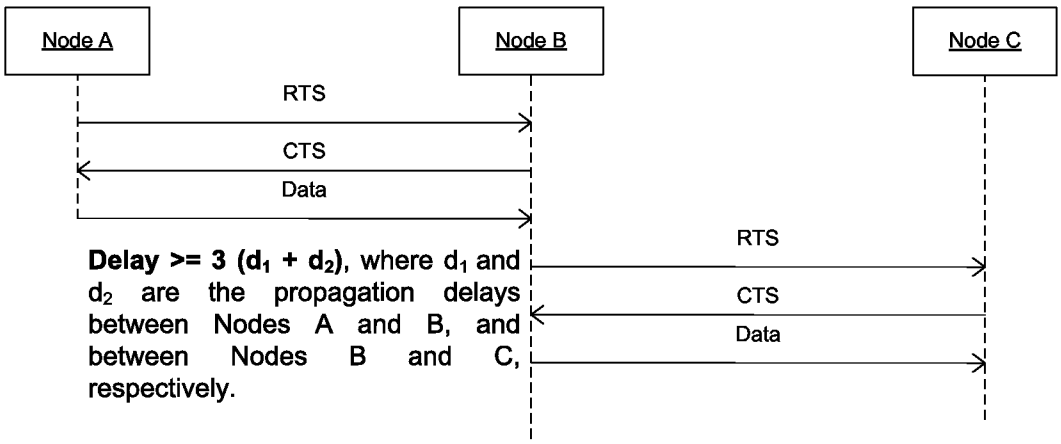

Figure 2. Message delay due to handshake protocol (recreated from [Xie 2001])

More over, with the current implementation, there is not a way to control the variability in message delay in a given session. For example, when supporting the communication of a mobile node, the transmission path lengths may vary. The propagation delay in UANs may be measured in tenths of a second and amplified by a factor of three for the control reservation messages, as described earlier. The resulting variance in message delays, also known as "jitter," limits the ability to support data sensitive to time sequence, such as video or an UUV feedback to navigation directives [Xie 2001].

A recent experiment by the Space and Naval Warfare Systems Command (SPAWAR) in San Diego, in conjunction with the Fleet Battle Experiment – India, examined the performance of the collision avoidance mechanism used to mediate access to shared acoustic channels by a collection of relay nodes. Just over 80% of the data packets reported by [Hartfield 2003], from a single-hop perspective, were exchanged

with only one RTS-CTS exchange, and another 10% were successfully exchanged after the second RTS-CTS exchange. Of the data messages examined, approximately 88% arrived without error, and another 9% were successfully received after a single retransmission. However, as it was pointed out in [Gibson 2005b], it may be the case that the chosen traffic pattern could be served as well by an uncontrolled MAC mechanism, as the lack of collisions for RTS-CTS frames seems to indicate.

2. *A Priori* Allocated Channels Approach

One proposed approach to overcome the performance limitations induced by large propagation delays in UANs is by the use of pre-allocated full-duplex channels between any two nodes in the reachable neighborhood, making the access to the media contention-free, thus eliminating the overhead of the CSMA/CA scheme. However, it remains to be seen whether or not this approach is capable of improving network performance, as it implies the division of the available bandwidth in mutually exclusive channels, thereby decreasing the already limited data rate for each individual channel. Additionally, due to the harsher physical environment in the ocean, this approach has met with some skepticism [Xie 2004]. However, the feasibility of full-duplex acoustic communications with experiments in a reverberation chamber has been shown. Several noise levels were tested, and the minimum signal to noise ratio level for which the message was still resolvable has been determined [Gibson 2002] [Xie 2004]. Although the feasibility of full-duplex communication in acoustic environments has been demonstrated, there still exists the need to demonstrate that this approach can improve network performance when compared with other MAC mechanisms, like the contention-based and the uncoordinated MAC approaches.

Another concern with this approach is related to traffic load considerations. If the sub-channels are insufficiently loaded, dividing the medium's available bandwidth into sub-channels may result in an increased message delay [Tannenbaum 2003]. That is, if the traffic pattern being served is such that the channels are idle a considerable amount of time, that bandwidth is wasted, while the effective transmission rate of an ongoing communication is decreased. If the bandwidth was not divided, the nodes that have traffic to send would have the full transmission capacity available, minimizing the transmission delay. Ongoing research is addressing this concern by applying techniques from related

domains, namely satellite and cellular systems. The goal is to develop a system that assigns dynamically, non-used channels to nodes that need to send data, and with that make an efficient use of the segmented bandwidth [Gibson 2005a].

In the next section we make a detailed review of MAC mechanisms that are most relevant in UANs, either because it is the approach being used in current implementations, the contention-based mechanism with collision avoidance, or because they constitute a viable alternative approach, thought to improve network performance, like the uncoordinated reactive or the *a priori* allocated channels MAC mechanisms.

B. MEDIA ACCESS CONTROL MECHANISMS

Contention management is important in helping to avoid or recover from collisions, and minimize the overall penalty in terms of wasted network resources in shared-medium environments, such as contention-bus or wireless networks. Typically, these methods include completely reactive, contention-based, and contention-free MAC mechanisms.

1. Reactive Media Access Mechanisms

Reactive MAC methods, similar to the Aloha protocol (developed by Abramson at the University of Hawaii for packet radio networks), provides no coordination mechanism prior to transmitting data. The system hosts simply transmit whenever they receive requests from their upper-layer applications and, then, wait for an acknowledgment from the addressed recipient. If no acknowledgment is received within a predefined period, retransmission of the unacknowledged data is performed, typically after a random exponential back-off period. The randomness is crucial here, in order to guarantee that after a collision the nodes that provoked the collision do not attempt to retransmit at the same time. Without this feature the access method would be useless for any appreciable number of hosts.

2. Contention-Based Media Access Control Mechanisms

While reactive access is a form of contention based access, no effort is made to limit the likelihood of collisions. More robust contention-based MAC methods typically provide a distributed coordinated access mechanism, such as the various Carrier Sense Multiple Access (CSMA) approaches. They seek to proactively avoid collisions, rather than only reacting to them, as the uncoordinated reactive MAC methods do. The CSMA

methods are further divided in Collision Detection (CD) as defined, for example, in [IEEE 802.3 2002], the Ethernet protocol standard for wired networks, and Collision Avoidance (CA) as defined, for example, in the protocol standard for wireless aerial radio-based networks [IEEE 802.11 1999]. At the center of the CA mechanism is the Network Allocation Vector (NAV) that is announced by the sending station in the transmitted frame. The NAV is the estimated duration that the current communication will take to finish. All other stations receiving the access coordination messages must take in consideration that estimation and defer their transmission attempts until their NAV timer expires. Upon the NAV expiration, the deferring nodes perform a random exponential back-off in order to minimize the chance that two deferring nodes attempt to transmit simultaneously.

The MAC mechanism in wireless networks challenges the CA framework used in wired networks because of the difficulty in detecting collisions. A collision can occur either while sending or while receiving. They have, however, different consequences. While sending, a node may experience a collision with an incoming frame from other node, if the transmission period of the outgoing frame overlaps the receiving period of the incoming frame. While transmitting, the node is unable to receive in the same channel. Typically, due to the difference in the signal strength, the outgoing signal will maintain its characteristics, and it will eventually get its recipient. The incoming frame however, is lost, and its sender needs to retransmit it [Kurose 2003].

After sending a frame, the sender is unable to determine whether a collision occurs at the receiver. A collision may occur at the receiver if the incoming two frames overlap their reception periods. If this is the case, both frames will be lost and both senders need to retransmit them. This latter situation is a consequence of the characteristics of the wireless channel. A third Node C may interfere with an ongoing communication between two other nodes, even if Node C (see Figure 3) is out of range of Node A. Two nodes may not detect each other due to either the fading of the signal's strength as it propagate through the wireless medium or physical impairments, and yet the signal may interfere with the communication of either of them with a third node (Node B in Figure 3) [Kurose 2003].

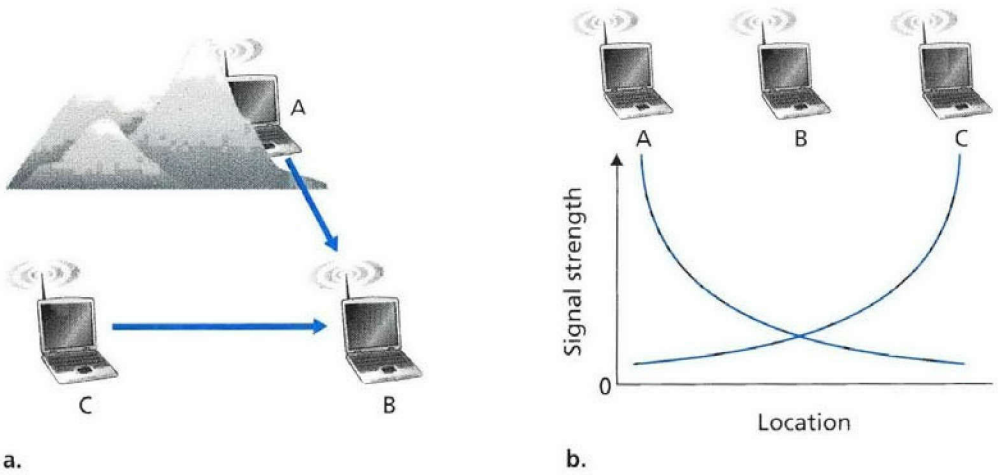

Figure 3. Hidden terminal problem (a) and fading (b) [Kurose 2003]

The 802.11 standard introduces, as an option, the possibility of exchanging short control messages to reserve access to the channel. They are defined as RTS and CTS control messages, and they attempt to reduce the contention period especially in the case when large data frames are pending transmission. After a successful RTS-CTS exchange, the source node has the channel reserved and can transmit the data frame. Without the RTS-CTS exchange, in the event of a collision, the need to retransmit the large data frames imposes a larger penalty on the network performance. With the RTS-CTS exchange, the possibility for collisions is reduced to the contention periods associated with the RTS-CTS exchange, and because the control messages are small, the penalty in the event of such collisions is much smaller [Gibson 2005b] [Kurose 2003].

The inability for the sender detect collisions at the recipient, and the hidden terminal problem (or fading), are issues that need to be addressed by the collision avoidance scheme in wireless aerial radio-based networks. These problems also need to be addressed in UANs, probably the reason why current UAN's implementation use some form of collision avoidance MAC mechanism.

3. Coordinated Contention-Free Media Access Control Mechanisms

There exists a group of MAC methods, which we can characterize as coordinated, contention-free mechanisms. These methods are considered alternatives to the CA schemes, and may be provided by moderated access via polling, token passing, or assigning each potential source with a dedicated transmission channel prior to sending

any data [Gibson 2005b]. The polling and token methods add additional overhead, due to the need to pass either the poll or the token, and they constitute a single point of failure that can affect the network. Usually, mechanisms for failure recovery and or fault tolerance can be implemented, with the inconvenience of adding complexity to the implementation and to network management.

In the case of dedicated channels, the required channels can be generated with different schemes, like Time Division Multiple Access (TDMA), Frequency Division Multiple Access (FDMA), or Code Division Multiple Access (CDMA). The dedicated channels provide for contention-free access to the media, thus avoiding the overhead of coordinating the access to the media. But it penalizes the frame transmission, because the available bandwidth was divided into discrete channels [Gibson 2005].

4. Media Access Control Mechanisms and UANs

Each of the MAC mechanisms has strengths and weaknesses. Those strengths and weaknesses are manifested when they are applied to a particular network topology, traffic pattern, or physical medium. For example, if we have a light traffic pattern with a small frame size, the uncoordinated access method may perform satisfactory, and even better than the other two methods. Under these conditions, the contention-based method may not overcome the threshold penalty, due to the overhead associated with the exchange of the two control messages.

A MAC method using *a priori* allocated channels may not overcome the transmission delay penalty induced by the reduced data rate that results from distributing the available bandwidth across the communicating nodes to establish the distinct channels. Moreover, with the typical UAN topology, where we have a backbone of relay nodes, the impact of queuing delay in the relay nodes may be significant. This happens because the sensor nodes associated with each relay node freely forwards their traffic, making the rate of arrival at the relay node possibly larger than the rate at which the relay node can serve each message. This may be exacerbated if each relay node does not have the full bandwidth available to forward its composite traffic, due to generating dedicated channels to eliminate contention. Additionally, as we consider the traffic being funneled towards the gateway, the relay nodes closer to the gateway will need to process, not only the messages generated within their sensor neighborhood, but also the messages

forwarded to them by the previous relay nodes in the path. This situation will be most critical near the gateway [Gibson 2005b].

C. COMPARISON AND EVALUATION OF MEDIA ACCESS CONTROL MECHANISMS IN UAN

An analysis and evaluation of different approaches to MAC in UANs is presented in [Gibson 2005b]. The study explored a very simple model, but provided some valuable insights on the trade-offs involved. The network studied (Figure 4) was a simple linear topology, where all the traffic is generated at Node 0, and terminates at Node 5. The intermediate nodes only forward the traffic from the previous node to the next node.

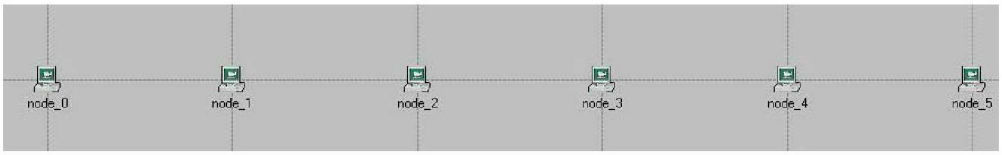

Figure 4. Simple six node linear trip-wire topology

Four cases were modeled. The first three cases considered variations on the contention-based with collision avoidance MAC mechanisms: frame switching with and without pipelining, and message switching (henceforth referred to as "CA/MS"). The fourth case to mediate the access to the medium considered the *a priori* allocated channels MAC scheme, which allows full-duplex communication.

In order to simplify the analysis and allow some early insights, the study made some strong assumptions: no errors requiring retransmissions were considered; all the details related with the physical layer, like regular propagation patterns, were ignored; of the partial delays that impact the overall network delay. Only the transmission and propagation delays were considered in the study. This level of abstraction allowed a significantly rewarding, first-order analysis that covered several aspects, like a comparison of the respective performance of the collision avoidance schemes, and the impact of hop distance, propagation rate, and the channelization degree [Gibson 2005b].

1. Evaluation and Comparison of Media Access Control Mechanisms

a. The Competing CA Variants

Of the three MAC mechanisms with CA, the CA/MS showed a better performance than the other two, especially when messages with multiple frames were

considered. The overhead of the RTS-CTS exchange in the CA/MS case is defined per message, and therefore with multiple frames per message more data is able to get through in a single RTS-CTS exchange. This was not the case for the other two CA's schemes, where an RTS_CTS exchange overhead was imposed for every frame, independently of whether or not they belong to the same message [Gibson 2005b].

b. Comparison of CA/MS with the a Priori Allocated Channels Scheme

The parameters considered (frame size, propagation distance, total message size, channelization, and number of hops) affect the latency of both schemes, but to different degrees, and the impact of different parameters may compensate for each other. For example, the impact of channelization (penalizing the *a priori* scheme) should not be considered by itself, because the hop count and the message size may compensate for the reduced transmission rate.

Three main ideas conditioned the comparison between these two schemes. The first is related with the intrinsic penalty that handshake-type protocols incur, imposing a three-fold increase in the overall total propagation delay, in an environment where the greatest component of the compounded delay is precisely the propagation delay. The second is related with the advantage that the a priori scheme would get when the specific parameter in consideration allowed the utilization of the pipeline effect. For example, large frame sizes benefit the CA/MS scheme. But if that large frame is broken in several small frames, then the *a priori* scheme can take advantage of the pipeline effect and outperform the CA mechanism. Finally, the main penalty of the *a priori* scheme is, as expected, the division of the available bandwidth in the sufficient number of channels to assure that each node have a unique channel within a two-hop neighborhood [Gibson 2005b].

The authors concluded that, when considering the MAC methodology, the implementor should consider the expected traffic load and the network topology. These factors include the number and size of frames being generated, the number of hops to traverse, the hop length, and the available transmission rate [Gibson 2005b].

D. OPEN ISSUES FOR FURTHER WORK

The first consequence of the conclusions expressed in [Gibson 2005b] is the fact that further study is only warranted for the CA/MS and *a priori* MAC methodologies. In

some cases, as we will be expanded latter, it may also be worth considering uncoordinated MAC mechanisms.

The referenced study made several recommendations, mostly regarding the relaxation of the assumptions in order to obtain results that are more general. Some of those recommendations will not be considered in this thesis, like the modeling of physical characteristics (namely the variance in physical parameters that have impact on the propagation and transmission characteristics of UANs). Another recommendation that will not be addressed is the exploration of dynamic capacity allocation in order to minimize the transmission penalty in the *a priori* scheme.

This thesis will, however, address three principal assumptions: the network topology, traffic generation in terms of variety and location, and the effects of physical layer induced errors (retransmission handling). Although the linear topology mimics some of the current UAN implementations, with the relaxation of the network topology assumption, we can consider other types of topologies, like a tree-type topology or a grid-type topology; i.e., topologies that may be more representative of current UANs implementations and more likely to impose greater demands on network resources.

Regarding traffic generation, the model should allow the introduction of other patterns of inter-arrival times, like exponential arrivals, representing a type of traffic generated by non-period, independent events. Additionally, traffic generation will be allowed in intermediary relay node neighborhoods, which will have an impact hard to predict in the overall delay of each scheme by means other than simulation. With CA/MS, the intermediary relay nodes will also need to compete for the media with its neighbor sensor nodes and, in addition, forward the traffic generated by them. The requests for the media will increase and that will have an impact on the overall end-to-end delay. In the *a priori* case, although the nodes do not need to compete for the media, the transmission rate will be affected by the channelization required to guarantee a unique channel for each node within its two-hop neighborhood. Additionally, a crucial factor in this scheme will be the queuing delay in the relay nodes, and the rate at which they will be able to forward the messages they receive from other relay.

Finally, it is still an open issue whether or not a simple Aloha-like MAC mode (henceforth referred to as "Aloha-like") is appropriate to serve networks with very low loads. This is the primary competing MAC mechanism to be considered in this thesis against the CA/MS. In order to evaluate this aspect, it is important to model some of the physical layer behaviors that affect the MAC mechanism, like the ability to detect collisions, both at the sender and at the receiver, and the ability to handle retransmission of data frames.

The following chapter describes the link-layer protocols modeled, one with the uncoordinated Aloha-alike MAC mechanism, and another with the contention-based with collision avoidance MAC mechanism with the message switch variant. The description encloses the protocols and their implementation in Opnet.

THIS PAGE INTENTIONALLY LEFT BLANK

III. PROTOCOL DESCRIPTIONS AND OPNET IMPLEMENTATION

A. PROTOCOL DESCRIPTIONS

The link-layer protocols of interest are the contention-based collision avoidance with message switching and the uncoordinated, Aloha-like, MAC mechanisms. The first is the better performing of the contention avoidance-based protocols referenced in [Gibson 2005b], and the second is a mechanism that has been looked as a possible solution for networks with low traffic demand.

1. Contention-Based Collision Avoidance with Message Switching (CA/MS)

This protocol is a derivative of the collision avoidance MAC used in radio-based networks.

a. Protocol Terminology

In order to facilitate the description of the protocol in the next sub-section, it follows the definition of some relevant terms of the carrier sense collision avoidance mechanism with message switching.

- Request-to-Send (RTS) – The RTS is the short frame used to request the channel. This message contains the duration that the node estimates to occupy the channel, and all the nodes that receive this message should take that in consideration and defer their transmission needs.
- Clear-to-Send (CTS) – The CTS is the short frame used as a response to an RTS. The CTS also contains the estimated duration that the channel will be occupied. The RTS-CTS exchange solves the hidden terminal problem (see Chapter II, Section B.2. for details about the hidden terminal problem). In the range of the two nodes, all neighbors are aware of their on-going communication.
- Data Frame Payload Size (DFPS) – The DFPS is the amount of the frame that will be actually used to transport the data bits.
- Acknowledgement (ACK) – ACK is the short frame used as response to acknowledge the reception of a data frame.
- Short Inter-Frame Space (SIFS) – There exists a non-zero delay between the time a stream of symbols is received at the physical receiver and the time it is available for layer two processing. More over, there exists a non-zero delay between the beginning of the layer-two processing and the time a response frame is ready to be sent. In order to account and abstract these delays, SIFS is defined as a time interval between frames that allow the

establishment of priority levels for access the wireless acoustic medium. It is closely related with the physical layer characteristics and technology. SIFS is used as the inter-frame spacing for response frames (CTS or ACK), and between data frames sent under the same RTS-CTS exchange.

- Distributed Inter-Frame Space (DIFS) – DIFS is the fixed period of time that the node needs to sense that the medium is free before transmitting a frame. This time is generally longer than SIFS. Therefore if a node only used SIFS, it will have an advantage in accessing the channel than nodes using DIFS. Not requiring DIFS in some nodes allows the prioritization of nodes in a network.
- Slot Time – Slot Time is a fixed period of time and is dependent upon the technology being used (for example, the maximum time the receiver takes to switch to transmitter mode), and the physical characteristics of the medium (for example, the propagation speed). The slot time is used as a unit in the back-off procedure. For example, a node performs a back-off of two slots, meaning that it will not attempt to transmit for the duration of two slots. This time must be counted during medium idle time.
- Back-off Window – The Back-off Window is the interval from which a number is randomly chosen to represent the number of slots to perform back-off.
- Contention Window – The Contention Window is the interval from which the upper bound of the back-off window is chosen. The first value chosen is the contention window lower bound and, for each unsuccessful attempt to transmit a data frame, the contention window upper bound increases exponentially, up to a maximum value.
- Retry Limit – The Retry Limit limits the number of unsuccessful attempts to transmit, after which the packet will be dropped.
- Buffer Size – The Buffer Size defines the size of the queue in which the incoming packets from the upper layer are queued. The queue may be filled up if the node cannot keep up with the rate of packets coming from the upper layer, and the node will start dropping packets.
- Timeout – Timeout is the period a sender waits for a response. If the Timeout period elapses, the node may perform a back-off, depending of whether it is the data originator. This description will be expanded when the back-off mechanism is explained further in the next sub-section.

b. Protocol Description

When a node needs to transmit, the node senses the medium for a DIFS period. If it is free after that period, it sends an RTS frame to its destination and waits. If the node receives the CTS within the timeout period, it will start forwarding the data frames. It sends all the data frames of the same packet, and the receiver, after receiving

all the frames, sends an ACK frame for that packet, with frame discrimination, (i.e., if it does not receive some of the data frames, it reports the reception of the ones successfully received, and the non-reception of the ones that were not received). This mechanism allows the retransmission of only the frames that were not received.

At the heart of this collision avoidance algorithm is the network allocation vector (NAV), which estimates the time a sender needs to occupy the channel. In the RTS frame, the sending node advertises the estimated time for transmitting and propagating all the data frames, including the time for receiving the acknowledgement. To calculate the expected maximum propagation delay, the node uses its own signal's maximum reachable distance. The receiving node advertises the same NAV when sending the CTS. After the RTS-CTS exchange, the nodes within range of the receiver that cannot be reached by the RTS of the sender should be reached by the CTS of the receiver (i.e., the mechanism to address the hidden terminal problem). Consequently, all the nodes in a reachable distance of both the sender and the receiver are aware that the medium is reserved for this communication.

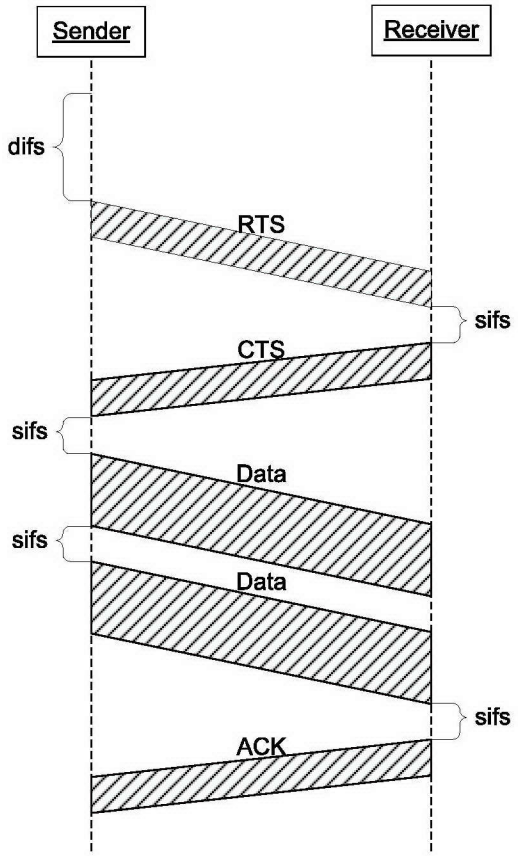

Figure 5. Collision Avoidance with Message Switching

A node is allowed to issue a CTS if it is a relay or a gateway node. They only issue CTSs if the received RTS was destined to it, and no ongoing communication between neighboring nodes is currently taking place (this is determined by checking whether its current NAV time is greater than the current time). After receiving the CTS, the sending node can estimate the actual distance to the receiver, and recalculates the NAV. Because the first NAV was calculated using the maximum reachable distance, this new NAV will always be less. The new NAV is advertised in the data frames and in the acknowledgement sent by the receiver. The neighbors of the sender and of the receiver update their NAV with the data frames and with the ACK frame. With this mechanism, all the reachable nodes of both the sender and the receiver will be aware of the new NAV. After a successful transmission of one message, the data-originator node performs a back-off.

A back-off is needed when the node preparing to initiate a medium reservation discovers that the medium is busy, or detects a collision. Additionally, when a data originator node infers a collision due to a timeout, it also needs to perform a back-off. Finally, after a successful transmission of a message, the data originator node performs a back-off. A node does not need to perform a back-off after a timeout if it is the data-recipient (e.g., waiting for data after sending a CTS). A data-recipient node will interrupt a back-off in progress to respond to a RTS, and completes the RTS-CTS exchange and data reception without interruption. After the end of that communication, it resumes the interrupted back-off. The back-off procedure only affects a node when it is the data-originator of an ongoing or future communication exchange.

A node performs a back-off for a specific number of slots, randomly chosen from a back-off window between zero and some upper bound. This upper bound is chosen from the interval defined by the contention window. The first value chosen for the back-off 's window upper bound is the minimum contention window value. However, for each additional back-off period, when attempting to transmit the same message, the back-off window's upper limit increases exponentially, up to the maximum contention window value. The back-off time accumulates only when the medium is idle. If the node senses the medium is busy, even if the frame is not sent to it, it interrupts the back-off period, and only resumes the back-off procedure after sensing the medium free (typically, when the NAV elapses).

After the completion of the exchange, a data-recipient node will either resume an interrupted back-off period, or immediately initiates an RTS-CTS exchange with the next relay node, ignoring any RTS from one of its neighbors. In UANs, this is typically the role of relay nodes. This design gives the traffic flowing in the backbone (i.e., traffic forwarded by relay nodes) a better chance to obtain access to a contested channel than the traffic from sensor or leaf nodes.

2. Aloha-like MAC Mechanism

In this mode, the node sends its data frames as soon as it receives a packet from the upper-layer protocol. It does not perform any back-off, nor provides NAV information in its transmission. The receiving node will receive and acknowledge the

data frame, if the frame did not collide at the receiver (i.e. when there is overlapping of the receiving periods of two or more frames at the destination location, or the receiver was transmitting).

Before sending the ACK frame, the receiver waits an SIFS (as defined in the previous section for the CA/MS protocol). Although the Aloha-like protocol does not require MAC coordination at layer-two, this allows the abstraction of the variable delays in the physical layer, and between the physical layer and the MAC layer to a single delay value, that happens to be the same to both protocols under evaluation.

The receiving node will not acknowledge a received data frame if it is waiting an ACK to a previously sent data frame. This design was chosen in order to remove the possibility of a collision between the transmitted ACK and the expected ACK. However, because this may penalize the protocol, this idea will be expanded in Chapter VIII.

If the source, does not receive an ACK because either the frame was not correctly delivered or the ACK was lost, the sender will timeout, wait a random period (back-off) and retransmit the frame. This protocol follows the stop-and-wait paradigm. That is, the source must receive an acknowledgement for each data frame before the next frame can be sent. In addition, after a successful frame transmission, the sender will perform a back-off, even if it has additional frames to send from the same packet or from a new packet. This protocol is also message switching, that is, the receiving node, usually a relay node, only forwards the packet after receiving all the frames of the packet. This means that the receiver must store the frames successfully received until the complete reception of the packet.

It should be noted that, whereas the Aloha-like mechanism performs a backoff between each successfully transmitted frames during a packet transmission, the CA/MS mechanism does not. This may be an undesired behavior of the Aloha-like protocol and will be discussed in Chapter VIII.

B. OPNET IMPLEMENTATION

The network modeling and simulation tool, OPNET Modeler, Educational Version, Release 10.5.A PL3, Build 2570 (henceforth referred to as "OpNet"), was chosen to implement the models of the network under study. OpNet is a simulation tool

primarily devoted to network simulation. It allows the analysis of a modeled system in terms of both behavior and performance by the use of Discrete Event Simulations to collect data. The suite incorporates tools for all phases of study, including model design, simulation, data collection, and data analysis [OpNet 2004a].

Basic network modeling in OpNet usually involves three stages: network modeling, node modeling, and process modeling. Network modeling is the stage where the general network topology is defined (for example, size, technologies, nodes, links, etc). Node modeling establishes the behavior of each network object defined in the network model, with the use of one or more modules connected with packet streams or statistical wires. The internal aspects of the modules will eventually determine the node behavior in terms of data creation or storage. Finally, the process modeling defines the underlying functionality of each one of the modules defined in the node model. It is represented by a Finite State Machine (FSM), and is created with icons that represent the states, and lines that represent the transitions between those states.

The following sections go more into the details of the network, node, and process models implemented in OpNet.

C. NETWORK MODEL

1. Topology

During the network modeling process, we define the network topology, network size, number of nodes, and the links that connect the nodes. In this thesis, we consider three topologies, the linear, tree, and grid topology. The linear topology is mainly considered for validation purposes through a control experiment with the use of a known test case. For this specific topology, we make additional assumptions in order to allow the validation by the analytical solution, like the absence of packet generation in the intermediary nodes. The other two topologies are considered in the simulation, and will be discussed in greater detail in Chapter IV.

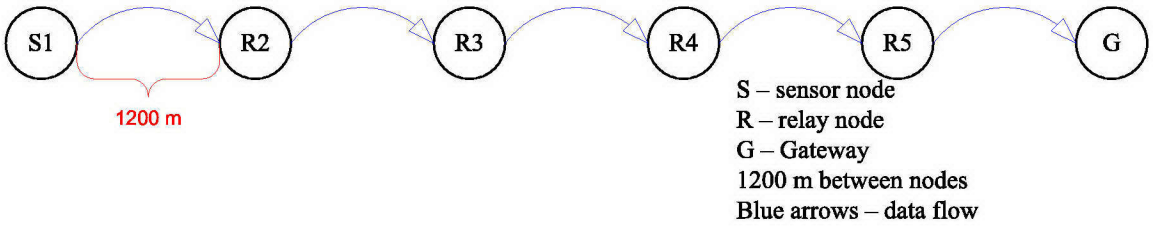

Figure 6. Network model with a linear topology

In the linear topology setting (shown in Figure 6), Sensor1 (S1) generates all the traffic, and the relay nodes only forward the traffic in a hop-by-hop fashion up to the gateway. The blue arrows represent the data flow in the network, depicted here only for illustration purposes. In each topology, the node name reflects its role (according to letter designation) and identification (i.e., number). In OpNet, the link between nodes is modeled at the node and process levels as a direct communication between nodes with the properly modeled delays. This will be covered in greater detail in the next section. The blue arrows represent the fixed routing, defined as a parameter of each node. Regarding the acoustic signal's range, also modeled as a parameter at the process model level, a sending node can only reach its 1-hop neighbors. For example, Relay3 can only reach the nodes, Relay2 and Relay4, regardless of whether sending an acknowledgement to Relay2 or a data frame to Relay4, as it should be in a reflexive wireless environment. Relay4 and Relay2 need to process that frame, determine if they are the correct recipient, and discard the frame if they are not.

2. Modeling Constraints

In order to model the wireless links, the communication initiated by a sending node should reach all nodes within its reachable neighborhood, regardless for which node the message is actually destined. This model did not account for physical layer behavior in terms of propagation patterns, error induced by the physical layer, or any other physical layer idiosyncrasies. We are only interested in modeling the behavior of the competing link layer protocols. For that reason, we needed to model the behavior that has direct impact on link layer performance. The propagation speed, the range of the acoustic signal, and the ability to determine collisions and handle retransmissions, are examples of interactions between the two layers that we need to account for when modeling the wireless acoustic link. However, even for these interactions, this study made several

assumptions to limit the complexity of the implementation. In doing this, a constant propagation speed and an idealistic propagation pattern that fades completely at a defined distance was assumed. Additionally, it was assumed that there was a global clock to which all the nodes were synchronized. Although the model accounts for errors in frames, it does not model the physical layer protocol behavior when frame errors occur.

3. Setting the Parameters

At the network level, all the parameters that are defined at the process model level are available for parameterization. When working at the Project Editor level, one has the layout of the network with all the nodes, and can access the parameters of each node through the context menu, accessible with the right-click of the mouse. In OpNet, it is also possible to provide the simulation run with a script file where those parameters can be defined. A more detailed description of the parameters and its default settings is made in Chapter IV.

Besides the object parameters referring to each node, the model has parameters that are defined at simulation time. These are considered simulation parameters and affect all the nodes. Typically, at least some of these simulation parameters could have more than one value, allowing multiple simulation runs with different parameters. In simulation terminology, these would be factors with several levels.

D. NODE MODEL

This study implemented one node model that can be configured to perform one of the two MAC mechanisms. Although the protocols have some differences, they are sufficiently alike to allow their modeling in the same node model and underlying process model, without adding too much complexity to the actual model.

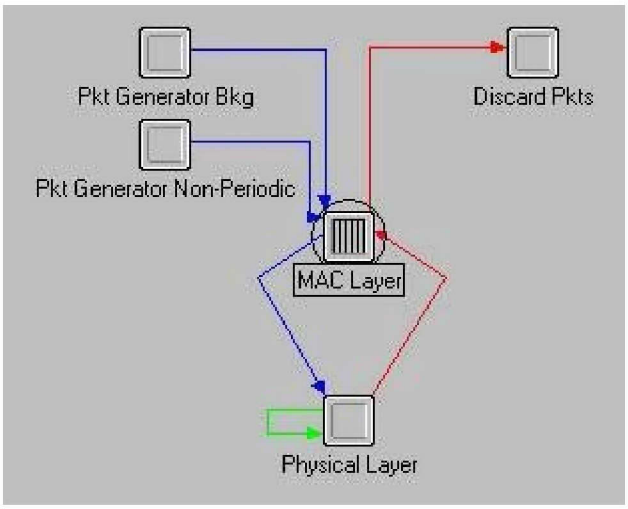

Figure 7. OpNet Node Model

The node is composed of five modules: the packet generator for the background traffic with a periodic pattern, the packet generator for the non-periodic traffic, the module for packet discard, the MAC Layer, and the Physical Layer. The first three modules have underlying process models that are part of the OpNet library. The packet generators allow the injection of different traffic patters according with the simulation requirements. In the present case, the background traffic is simulating the constant-rate (periodic) traffic generated by data gathering sensors in the network, and the non-periodic traffic is simulating traffic generated by some discrete event (a ship passing by, for example) that generates a special report. As stated in Chapter I, this study wanted to evaluate how the network can serve this non-periodic traffic, when it is already serving the regular traffic, in terms of end-to-end delay. The module for discarding packets is simulating the upper layer to which the MAC Layer at the Gateway forwards the received packets.

The MAC Layer is based on a modification of the OpNet model of a radio-based wireless network, with the necessary changes to meet the required behavior of UAN's nodes (i.e., sensor, relay, and gateway nodes). The Physical Layer module is a process model, specifically made for this study's design. As already mentioned, the model does not address individual physical layer characteristics. The details of the Physical Layer model are discussed in the next section.

The lines connecting the modules are called OpNet streams. They indicate the flow of packets between modules. For a specific module, there are incoming and outgoing streams, indicated by the direction of the arrow. The Physical Layer module has one stream originating and ending on itself (see the green arrow on Figure 7). This artifact allows the node to receive packets from other nodes in that incoming stream. The sending node creates a remote event in the receiving node and delivers the packet on this incoming stream. The receiver then extracts the incoming packets. The next section will describe, in detail, the process model underlying the MAC Layer and the Physical Layer.

E. PROCESS MODELS

The process model is where one defines the intended behavior of the modules defined at the node level. This is accomplished with the combination of a FSM, with states and transitions, and C-language code, defining the set of actions in each state.

1. The Physical Layer Process Model

The Physical Layer process model determines the reachable nodes and sends the frames received from the MAC Layer to all of them. After receiving a frame from a remote node the model does not perform any screening on it. It forwards the frame to the MAC Layer. It is the MAC Layer responsibility to determine whether or not the frame is destined to the current node.

The Physical Layer process model implements the behavior of the Physical Layer module at the node level. It encompasses five states with the respective transitions (Figure 8). The red areas represent unforced states; that is, after the execution of the "enter executives" the module releases the control to the kernel. When the kernel has events for the current module to execute, then the module resumes, executing the "exit executives" of that state, evaluate the transitions and then proceeds in the path that has a true transition. In the case of the green states, which represent forced states, the execution is not released to the kernel, all the C-code of the state is executed, the transitions of the state are evaluated and the execution will follow the path of the true transition. The execution will not be released to the kernel until the path of execution encounters an unforced state. The lines between the states represent the transitions. After executing the "exit executives" of some state, the transitions are evaluated and only one of them can and should be true. The transitions may depend on variables changed during the state

code execution, but they are independently defined. The *default* transition is a way of establishing that the state can accept any other interrupts.

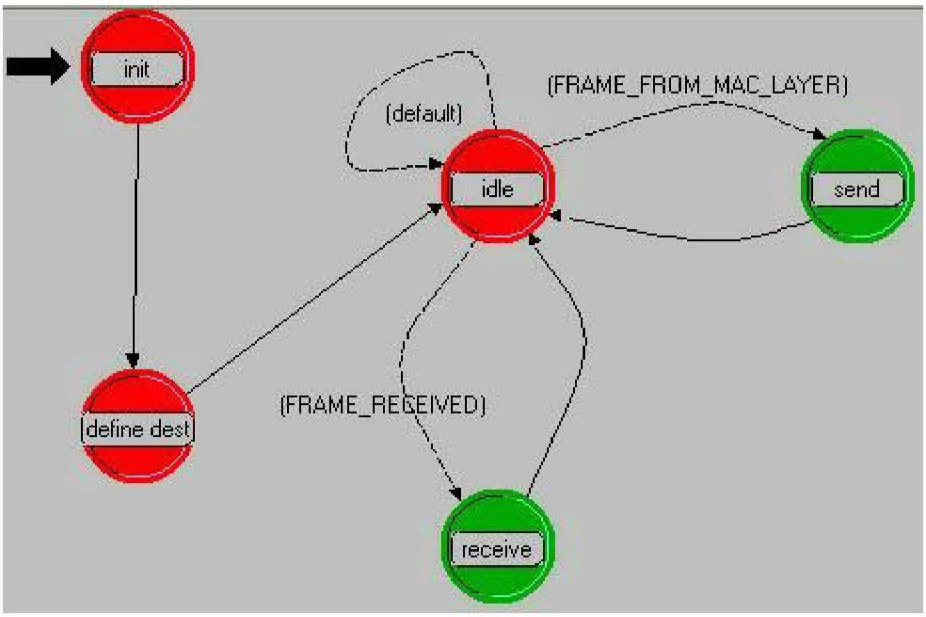

Figure 8. Physical Layer process model

The Physical Layer process model has five states (Figure 8). The *init* state initializes the state variables and reads the required attributes. In the *define_dest* state, the module creates a list with all the reachable nodes. In the *idle* state, the module waits for either a frame from the MAC Layer to send to the reachable nodes, or a frame that a remote node sent to the current node.

When the module receives a frame from the MAC Layer, it changes to the send state (a forced state), performs the transmission of the frame to all reachable nodes and returns to idle waiting for another frame. The module sends the frame creating a remote event in the green stream depicted in Figure 7. When the node receives a frame from a remote node, it receives a stream interrupt, changes to the receive state, retrieves the frame from the stream, sends it to the MAC Layer, and returns to idle waiting for additional frames.

The overall event generation is described in Figure 9 below:

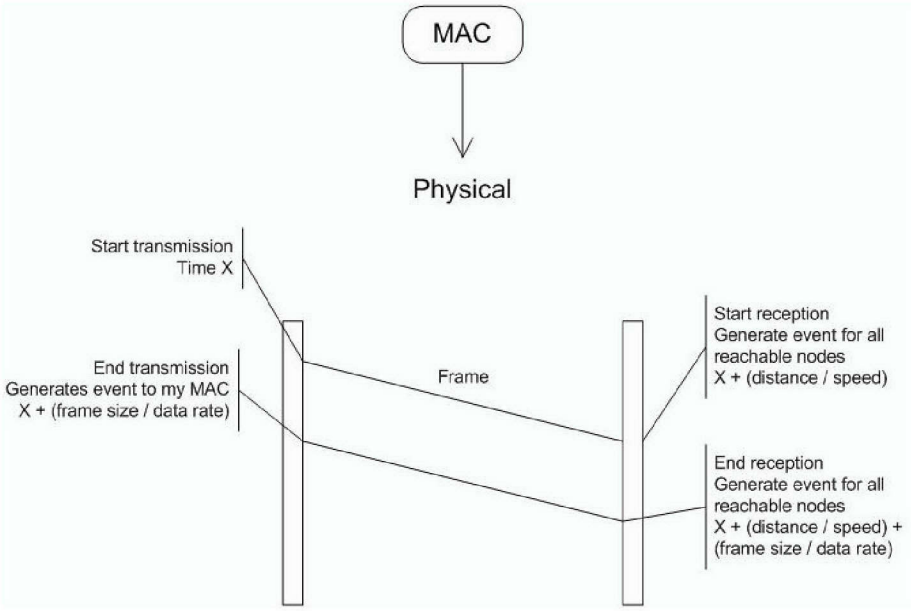

Figure 9. Physical interrupts modeling generation

The time, X, is defined as the time when the module receives the frame from the MAC Layer. The end transmission, start reception, and end reception events are all generated at the same time, and will be executed in accordance with the transmission and propagation delays calculated by the module. The order at which this process occurs is very important in order for the receiving nodes to be able to determine the occurrence of collisions. The actual sending of the frame is not represented in Figure 9. The frame is set to be received at the same time as the end of reception event, but because it is scheduled first and the kernel maintains that order, the actual reception of the frame is before the end of reception event, as it should be.

2. The MAC Layer Process Model

a. General Description

The main function of the MAC Layer is to establish the conditions under which the node accesses the medium. In the case of CA/MS MAC mechanism, it needs to defer access and execute a RTS-CTS exchange. On the other hand, the uncoordinated Aloha-like MAC mechanism does not defer access and sends the data frame without any prior coordination.

The MAC process model implements the behavior of the MAC Layer module at the node level. It encompasses seven states with the respective transitions (Figure 10).

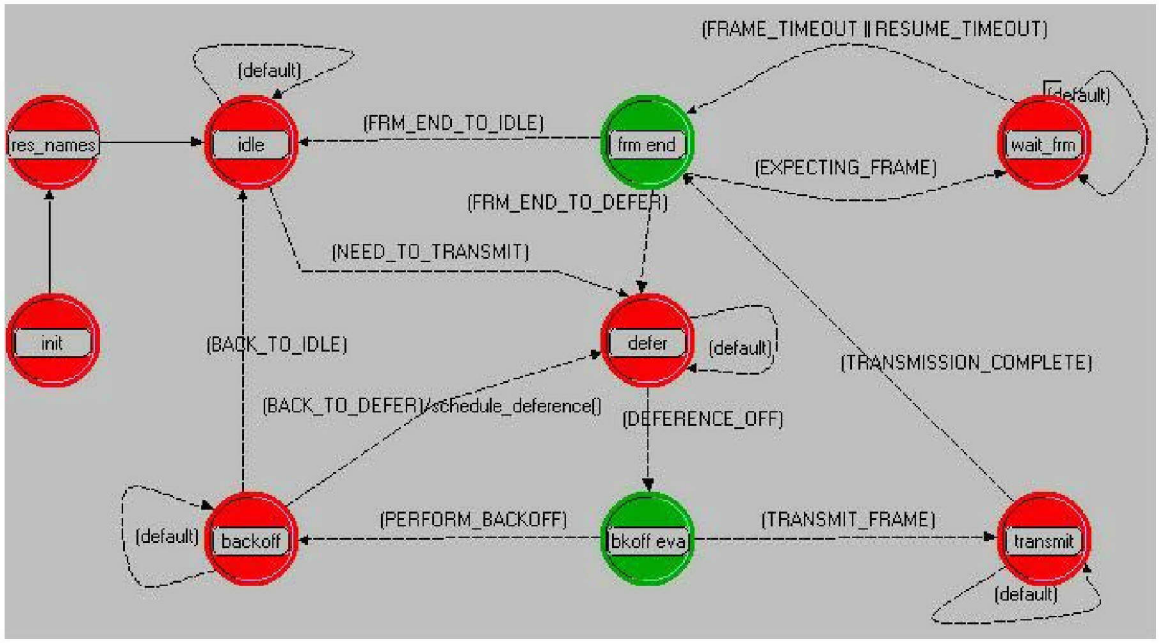

Figure 10. MAC process model implemented in OpNet

The *default* transition is a way of establishing that the state can accept any other interrupts. The default transition in needed because, as will be shown later, all the states can, for example, accept interrupts of incoming packets from the physical layer. In the following subsections, the states will be described in more detail.

b. Event Table

The list of all the logical events that may occur during the process is described in the next table. For each event the event implementation method (interrupt type) is defined.

Event name	Event description	Interrupt type
Power up	Initialization	Begsim
Packet arrival	A packet arrived from the upper layer	Stream
Frame Arrival	A frame arrived from the lower layer	Stream
Receiver on	The node started receiving a frame	Remote
Receiver off	The node ended a frame reception	Remote
Deferment off	The deferment timer has expired	Self
Back-off end	The back-off period has ended	Self
Timeout	The expected frame was not received	Self
Resume timeout	The expected frame was received on time	Self
Transmitter off	The node finished transmitting a frame	Remote
Unnamed	Unnamed generic event to provide the initial state transitions	Self
End simulation	The criteria for ending the simulation was reached	Endsim

Table 1. MAC Process Model: Event Enumeration

c. States Description

Considering the events defined in Table 1, we can describe which events are actually accepted by each state:

State	Events accepted by the state
init	Begsim
res_name	unnamed
idle	packet arrival, frame arrival, receiver on, receiver off, end simulation
defer	packet arrival, frame arrival, receiver on, receiver off, deference off, end simulation
back-off	packet arrival, frame arrival, receiver on, back-off ended, end simulation
transmit	packet arrival, frame arrival, receiver on, receiver off, transmitter off
wait_frm	packet arrival, frame arrival, timeout, resume timeout, receiver on, receiver off, end simulation

Table 2. MAC Process Model: Event Acceptance Table

Each state has a set of tasks that should be accomplished in order to maintain the overall consistency of the model. A generic description of those tasks, by state, follows:

State	Tasks
init	This state initializes the state variables, reads the models attributes, creates global lists, and registers the statistics handlers.
res_names	This is an auxiliary state to give time to each node, in the beginning of simulation, to identify the name of the node to which it should forward its data. This could not be done in the init state because the node name is an attribute that is read when the simulation begins and only after the execution of the enter executives of the init state of all the nodes in the network is that attribute known by each node. Thus, when the first node transits to the *res_names* state all the nodes have already read their own attributes. The name of the destination node is needed only for OpNet programmatic idiosyncrasies;
idle	The purpose of this state is to wait until a packet has arrived from the higher or lower layers. All the processing regarding the arrival of packets from the higher layer, frames from the physical layer, and determination of whether a collision occurred is handled by a special function called *interrupts_process*, which we will describe later. If the node has a packet or a frame to send then it schedules a deferment timer in accordance with the logic depicted in the function *schedule_deferment*, which we also describe later.
defer	Together with the *schedule_deferment* function, this state controls the logic for deferment. In this state, the node defers until the medium is available, in accordance with the rules defined for each protocol
bkoff_eva	in this state the node evaluates whether a back-off is required for the frame the node is trying to transmit.
transmit	In this state if it receives a packet from the higher layer it queues the packet without changing the state. After the transmission is completed, it changes the state to the *frm_end* state. Any frame received from the physical layer during this state will be considered a bad frame and discarded.
back-off	In this state the random back-off is processed. If a packet from the higher layer is received then it queues it without changing state. If the receiver becomes active it interrupts the back-off procedure and changes the state to *defer*. After ending the back-off period, if the node has packets to send then it changes state to *defer*, otherwise it changes state to *idle*

frm_end	The purpose of this state is to determine the next unforced state after completing the transmission. In this state, the behavior of the two considered protocols are different: • If expecting a frame set timeout and change to *wait_frm* state • If a frame is not expected, then change state to defer or idle depending on whether or not the node has packets or fragments to send. • In the case of CA/MS MAC protocol: *if RTS is transmitted then wait for CTS* *if CTS is transmitted then wait for DATA* *if all the data fragments were transmitted then wait for an ACK* • In the case of Aloha-like MAC protocol: *if DATA is transmitted then wait for an ACK*
wait_frm	The purpose of this state is to wait for a response after transmitting a frame that requires a reply. After receiving the expected frame reset the timeout, set the expected frame type to none and change state to *frm_end*. Depending on the modeled behavior for each protocol, if this corresponds to the end of packet or frame transmission, the node needs to back-off. If the expected frame was not received, then the node needs to perform a back-off, aggravated by the number of retransmission attempts.

Table 3. MAC Process Model: Generic Tasks Performed by Each State

d. State Variables

The state variables have a crucial role on defining the current state and the overall behavior of the node model. Practically all of the transitions are defined as a function of some of the state variables, and therefore, before we provide the description of the state transitions it is advisable to describe the state variable implicit in those descriptions. The following is a definition of the state variables with most influence the behavior of the module:

OpNet Name	Description
backoff_required	This flag is on when a back-off is required
collision	Set when the reception period of two or more frames at a single node overlap each other. The frames received during this period are considered bad and discarded.
expectedFrameType	Variable that holds the frame type that the node expects after sending a frame. It can assume the enumerated types described in the previous description. The UanE_None_Transit is a special type that allows a more effective control of the state of both the sender and the receiver. For example, after receiving a RTS frame, the node, assuming that it complies with the requirements for CTS issuance, needs to send a CTS. Without this enumerated type, the node will remain with the expected frame type equal to UanE_None, and if, before sending the CTS it received another RTS from another node it would try to respond to it, as the general case, with a CTS frame. This is not an intended behavior because the node is already committed to an ongoing communication with the first RTS. Therefore, the type UanE_None_Transit signals a behavior where the node is still not yet expecting a frame, but it is already committed to an ongoing communication.
fragments_to_send	One at a time a packet is taken from the higher layer queue and put into the fragmentation transmission buffer. This flag is on as long as the fragmentation buffer is not empty.
frameTypeToSend	Variable that holds the defined frame type to send. It can assume the following enumerated types: UanE_None, UanE_Rts, UanE_Cts, UanE_Data, UanE_Ack, UanE_Rts_Ime, and UanE_None_Transit. The first five are self-explanatory, indicating the type of frame that the node needs to transmit, whether as a response or determined by the need to send a data packet. The remaining two enumerations are use in the CA/MS MAC protocol. The UanE_Rts_Ime is a special enumeration to allow a relay node, after receiving a data packet, to transmit immediately after the expiration of the navTime and the receiverIdleTime + difsDuration. The alternative would be to use the UanE_Rts to content for the channel. However, at the time it tries to content for the channel, immediately after sending the Ack, the navTime would not have expired yet and, as in the general case, it should perform a back-off. To correct this behavior the UanE_Rts_Ime was introduced. The UanE_None_Transit is used for the state variable, expectedFrameType, described next.
nav_updated	Set every time the node's NAV is updated. It indicates the need for calculating a new deferring timer.
navTime	Variable that holds the node's NAV in absolute simulation time.

packet_to_send	This flag is set to true when the queue that holds the incoming packets from the higher layer is not empty.
perform_backoff	This flag is on when the node needs to perform a back-off immediately.
rcvd_bad_packet	When a node is transmitting, any frame that is received at that node is classified as bad and discarded.
receiver_busy	Set during the period where the receiver is busy.
receiverIdleTime	Absolute simulation time when the received idles.
rts_sent	A sender node sets this flag after a successful RTS-CTS exchange.
transmitter_busy	Set during the period where the transmitter is busy.

Table 4. MAC Process Model: State Variables Description

e. Interrupt Codes

The interrupt types also play a crucial role in defining the overall behavior of the module and the state transitions, in particular. They allow the identification of user-defined interrupt codes that are used in some of the state transitions, thus affecting the flow of execution. The following is the description of the user-defined interrupt codes used:

- UanE_Deference_Off – identifies the current interrupt as an end of a deference interrupt. The node should end the deference period.
- UanE_Frame_Timeout – the maximum wait time for the expected frame was reached. The node should stop waiting for the frame.
- UanE_Resume_Timeout – the expected frame was received, the node assumes a successful transmission, and therefore the timeout interrupt should be erased.
- UanE_Back-off_Elapsed – the end of the back-off period was reached.
- UanE_Transmitter_On – remote interrupt from the Physical Layer stating that the transmitter is on. This interrupt is not really used because the MAC Layer sets the *transmitter_busy* flag when it sends the frame to the lower layer.
- UanE_Transmitter_Off – remote interrupt from the Physical Layer stating that the transmitter changed to off state (ended transmitting a frame).
- UanE_Receiver_On – remote interrupt from the Physical Layer stating that the receiver changed to the on state (started receiving a frame).
- UanE_Receiver_Off – remote interrupt from the Physical Layer stating that the receiver changed to the off state (ended receiving a frame).
- receiverInterruptsSemaphore – variable that keeps track of the number of on and off interrupts received from the receiver (Physical Layer). An "on" interrupt adds one, and an "off" interrupt subtracts one. The receiver is

idle when this variable is zero. If the semaphore is bigger than one it means the node is experience a collision. After experiencing a collision, the semaphore needs to go to zero before the collision flag can be set to false. That is, each frame being received must complete "reception" before the collision condition can be cleared. All the frames received with the collision flag set are discarded.

f. Interrupt Stream Codes

There also three additional interrupts of type stream that have influence upon the overall behavior of the module. Their descriptions are as follows:

- INPUT_STREAM_FROM_UPPER_LAYER_BACKGROUND – code that identifies the interrupt stream as an interrupt stating that a packet from the upper layer packet generator with the generation characteristics of background (periodic) traffic has arrived and is ready to be picked.
- INPUT_STREAM_FROM_UPPER_LAYER_NON_PERIODIC – code that identifies the interrupt stream as an interrupt stating that a packet from the upper layer packet generator with the generation characteristics of non-periodic traffic has arrived and is ready to be picked.
- INPUT_STREAM_FROM_PHYSICAL_LAYER – code that identifies the interrupt stream as an interrupt stating that a frame from the Physical Layer has arrived and is ready to be extracted.

g. State Transitions

With the relevant state variables (Table 4) and interrupt codes described, auxiliary macros are defined in Table 5 and, finally, in Table 6 the state transitions are described.

OpNet Macro	**Definition**
FRAME_RCVD	UanE_Receiver_On && !bad_packet_rcvd
BACKOFF_COMPLETE	UanE_Back-off_Elapsed && !receiver_busy
FRAME_TO_TRANSMIT	packet_to_send || fragments_to_send || frameTypeToSend != UanE_None || backoff_required

Table 5. MAC Process Model: Auxiliary macro definitions

State	Transition	New state	Description / definition
idle	NEED_TO_TRANSMIT	defer	packet_to_send || fragments_to_send || frameTypeToSend != UanE_None
defer	DEFERENCE_OFF	bkoff_eva	UanE_Deference_Off && !receiver_busy
bkoff_eva	PERFORM_BACKOFF	backoff	perform_backoff
bkoff_eva	TRANSMIT_FRAME	transmit	!PERFORM_BACKOFF
backoff	BACK_TO_DEFER	defer	FRAME_RCVD || (BACKOFF_COMPLETED && NEED_TO_TRANSMIT)
backoff	BACK_TO_IDLE	idle	BACKOFF_COMPLETED && !NEED_TO_TRANSMIT
transmit	TRANSMISSION_ COMPLETE	frm_end	UanE_Transmitter_Off
frm_end	EXPECTING_FRAME	wait_frm	expectedFrameType != UanE_None && != UanE_None_Transit
frm_end	FRM_END_TO_DEFER	defer	!EXPECTING_FRAME && FRAME_TO_TRANSMIT
frm_end	FRM_END_TO_IDLE	idle	!EXPECTING_FRAME && !FRAME_TO_TRANSMIT
wait_frm	FRAME_TIMEOUT || RESUME_TIMEOUT	frm_end	UanE_Frame_Timeout || UanE_Resume_Timeout

Table 6. MAC Process Model: State Transitions Definition

h. Relevant Functions for the Process Model Flow

Some functions are very important for the control of the process model flow. In the following paragraphs, the main logic defined in each of them is described.

1. The *interrupts_process* function is called in all states to process the interrupts from the streams (upper and lower layers), and the interrupts from the receiver. The main logic in the function is as follows:

```
if (interrupt stream from upper layer) then
        retrieve the packet from the stream and put it in the queue
        set packet_to_send true
if (interrupt stream from physical layer) then
        process a physical layer arrival
if (receiver On) then
        if (receiver busy) then
```

```
                    set collision to true
                set receiver busy to true
                add one to the receiver semaphore
        if (receiver Off) then
                subtract one to the receiver semaphore
                if (receiver semaphore == 0) then
                        set collision and receiver busy to false
                        set receiver idle time to current time
        if ((packet to send || fragments to send) && !rts_sent &&
            expectedFrameType == UnaE_None &&
            frameTypeToSend == UanE_None)) then
        set frameTypeToSend = UanE_Rts
```

2. The *schedule_deferment* function is another function with great influence in the behavior, specially, of the CA/MS MAC protocol. The main logic in the function is as follows:

```
if (back-off is required && frame type to send is neither UanE_Cts nor
   UanE_Ack) then
        set perform back-off to true
        if (the frame type to send is UanE_Rts_Ime) then
                set frame type to send UanE_Rts
        if (network mode is Aloha-like) then
                set end of deference to current time
        else
                set end of deference to NAV

else if (network mode is Aloha-like) then
        if (frame type to send is UanE_Rts)
                set end of deference to current time
        else
                if (frame type to send is UanE_Rts_Ime) then
                        set frame type to send to UanE_Rts
                set end of deference to current time plus sifs

else if (frame type to send is UanE_Rts_Ime) then
        set frame type to send as UanE_Rts
        set the end of deference to the biggest of NAV and
                                        (receiverIdleTime plus difs)

else if (frame type to send is UanE_Rts) then
        if (MEDIUM_IS_IDLE)
                set end of deference to current time
        else
                set back-off required to true
                set perform back-off to true
```

set end of deference to NAV

else if (frame type to send is not UanE_None)
set deference to current time plus sifs

F. ADDITIONAL DETAILS

Additional details about the implementation can be seen in Appendix A and B, where the C-language code of the OpNet implementation is presented.

This chapter described the two MAC protocols, the CA/MS and the Aloha-like, and their implementation in OpNet. In the next chapter, the simulation design will be described.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. SIMULATION DESIGN

A. SYSTEM DEFINITION

The goal of this thesis is to study and compare the performance of two different MAC mechanisms in an underwater acoustic networking environment. The key component of this study is the chosen MAC protocol, implemented at the link layer. Therefore, the system under study is the network link layer.

B. SERVICES

Two MAC protocols are investigated in the system under study. The first protocol accesses the medium based on distributed contention with collision avoidance (CA/MS), and the second protocol uses an uncoordinated Aloha-like mechanism to access the medium. Both provide message switching.

It is assumed that both services offer message deliver free of errors. They also offer two different traffic patterns at the same time: one based in a constant rate generation emulating a data gathering network and is considered the *background* traffic, and the other based in a non-periodic rate generation emulating the need for communicating the occurrence of some non-predictable event and is considered the superimposed *non-periodic* traffic.

C. METRICS

For each type of service, the end-to-end delay and throughput will be compared. Additionally, the performance of the network link layer will be compared when subject to the different traffic patterns. This leads to the following performance metrics:

1. End-to-End Delay

The end-to-end delay is the time that a message takes to traverse the network, from the point at it was delivered by the upper layer to the network link layer at the source node, to the point at which it will be delivered to the upper layer again at the destination node (in UANs usually a gateway). Each simulation run calculates the average of the end-to-end delay of all the messages that arrived to their final destination. This can be expressed in the following expression:

$$Average\ End-to-End\ Delay = \frac{\sum\left(Time_{message\ delivered\ to\ upper\ layer} - Time_{message\ delivered\ by\ upper\ layer}\right)}{\sum messages_{received\ by\ upper\ layer}}$$

The desired performance results from the minimization of the average end-to-end delay.

2. Throughput

The throughput is the rate at which data units are delivered to the upper layer, usually expressed in bits per second. Throughput is a measure of network utilization and the maximum throughput represents the available network capacity. Each simulation run calculates the rate at which the data units were delivered to the upper layer. This can be expressed in the following expression:

$$Throughput = \frac{\sum bits_{delivered\ to\ upper\ layer}}{time_{simulation\ duration}}$$

The desired performance results from the maximization of the observed throughput.

D. PARAMETERS

The traffic generation in OpNet has two relevant parameters that determine the traffic characteristics: the *packet size* and the *inter-arrival time*. There are additional parameters, like an on and off periods of the generation pattern, but because we set the on period to be equal to the simulation time and the off period to zero, they do not play any role in the simulation. The *packet size* is considered a fixed parameter, and will be described in this section, and the *inter-arrival time* is considered a factor, or a variable parameter, and will be discussed in the next section.

The parameters, described later in this section (Table 7), although defined as model attributes in different process models during their design, are available in the Project Editor at network level. The process used to define them was the Graphical User Interface, accessible through the context menu, activated by the mouse right-click over the node of interest, then selecting the command *edit attributes*.

The parameters that affect the performance of the system and the values to which they were set are presented in Table 7, together with the correspondent OpNet variable name:

Descriptive Name	**OpNet Variable Name**	**Default Value**
Contention Window Low Bound	`minContentionWindow`	5
Contention Window High Bound	`maxContentionWindow`	20
DIFS	`difsDuration`	1.08 s
SIFS	`sifsDuration`	0.01 s
Slot	`slotDuration`	1.08 s
Size of the Higher Layer Packet Arrival Buffer	`highLayerListMaxSize`	256,000 bits
Number of Retransmission Attempts Allowed	`retryLimit`	7
Size of ACK frame	`sizeACK`	80 bits
Size of RTS frame	`sizeRTS`	80 bits
Size of CTS frame	`sizeCTS`	80 bits
Size of Data Frame Header	`sizeDataFrameHeader`	80 bits
Frame Error Rate	`errorRate`	0 %
Propagation Speed	`propagationSpeed`	1500 m/s
Data Rate	`dataRate`	1000 bits/s
Range	`range`	1500 m
Message Size of the Background Traffic	`packet_size_string`	1024 bits *a)*
Destination Node's Address	`destinationNode`	*b)*
MAC Address	`myAddress`	*b)*
Type of Node	`typeOfNode`	*b)*

Table 7. Process Model Parameters

a) This attribute belongs to the Process Model of the Background Traffic Packet Generator.

b) This parameter is defined in the network model between three types, sensor, relay and gateway nodes, and once selected should remain fixed throughout the simulation runs.

E. FACTORS (VARIABLE PARAMETERS)

The factors, described later in this section (Table 8), although defined as global attributes of different process models during their design, are available in the Project Editor at simulation time, that is, when the simulation is first set up. At the Project Editor level, after activating the *Configure/Run Discrete Event Simulation (DES)* command, a Graphical User Interface pop-up with several selections to set the simulation run(s). The factors are accessible selecting in the left tree the options *Inputs*, and then *Global Attributes*. With this operation, the factors are available in the working window and they can be set to the levels of interest.

The key factors chosen for this study are listed in Table 8, with the correspondent OpNet variable name:

		Level	
Descriptive Name	**OpNet Variable Name**	**Nr**	**Enumeration**
Network Topology	Not Applicable	2	Tree | Grid
Network Link Layer Mode	`networkMode`	2	Aloha | CA/MS
Data Frame Payload Size	`dataFramePayloadSize`	4	128 | 256 | 512 | 1024
Background Traffic Inter-Arrival Time	`intarrvl_rate_string`	10	Constant distribution with mean: 48, 65, 80, 102, 120, 144, 180, 240, 360, and 720 *a)*
Non-Periodic Traffic Generator	Not Applicable	2	None | Active *b)*

Table 8. Process Model Simulation Factors (Variable Parameters)

a) This attribute belongs to the Process Model of the Background Traffic Packet Generator. The levels for this factor were chosen after some earlier experiments to determine the range of values that would yield sensitivity results.

b) The active mode of the non-periodic traffic pattern will only be chosen for a new set of simulation runs with the data frame payload size that showed the best performance.

With the introduction of non-periodic traffic, one is interested in evaluating the performance metrics of four or five occurrences during the simulation time. In addition, the typical message size is bigger than the one generated by the constant pattern. The relevant characteristics for defining the generation are the *packet size* and the *inter-arrival time*. As in the case of the background traffic, the on and off periods of the

generation pattern are irrelevant, because the on period was set to equal the simulation time and the off period equal to zero. Therefore, when selected to active, the non-periodic traffic generator has the following settings:

Descriptive Name	OpNet Variable Name	Default Value
Packet Size	packet_size_string	5120 bits *a)*
Inter-arrival Time	intarrvl_rate_string	exponential(600) *a)*

Table 9. Non-Periodic Traffic Pattern Generation

a) Both attributes belong to the Process Model Non-Periodic Traffic Packet Generator.

F. DESIGN

A full factorial design with the restrictions noted in the last section is chosen. Therefore, with the two levels in Network Topology, two levels in Network Link Layer Mode, four levels in Data Frame Payload Size, and ten levels in the background traffic inter-arrival time, yields a total of 160 simulation runs. Initially, the Non-Periodic Traffic will be set to none in the 160 simulation runs. Of the four data frame payload sizes, the one that showed the best performance in the previous runs will be used in an additional set of simulation runs (Network Topology with two levels, Network Link Layer Mode with two levels, Background Traffic with ten levels) with the Non-Periodic Traffic set to active. This yields an additional forty simulations runs, totaling 200 simulation runs.

G. SIMULATION TIME AND RANDOMNESS

The determination of the required simulation time is usually driven by the requirement of ensuring that a sufficient amount of time has elapsed to allow representative activity to occur in the simulated model. Earlier simulations runs of one hour showed that the values for end-to-end delay and throughput already achieved a stable value or trend at the end of simulation. Therefore, a simulation time of one hour was chosen for the 200 simulation runs.

The simulation model implemented involves randomness when nodes perform backoff a random amount of slots. Therefore, the results are subject to fluctuation, and when a single result is considered, the only assertion that one can make is that the result is possible. With a single result one cannot draw conclusions regarding the trend

exhibited in the results; neither compares the performance of the two MAC protocols in a statistically significant manner.

To build confidence on the obtained results, one needs to make several simulation runs with the same network setup, and with different seeds for the random number generator. This is a common procedure used to take advantage of the Central Limit Theorem, in order to allow going from working with an unknown distribution (e.g., the random variable in question, such as the end-to-end delay), to a situation where we work with the sample mean of that random variable. The sample mean, which is also a random variable, according to the theorem, follows a known distribution, such as a normal distribution. In the case that the standard deviation of the original random variable is not known, as long as the sample size is sufficiently large (usually bigger than or equal to thirty), the sample variance can be used in place of the true variance to compute the confidence intervals. For computing confidence intervals, the OpNet Analysis Tool uses a t-student distribution for sample sizes less than thirty and a normal distribution for samples sizes greater than or equal to thirty. Therefore, for each of the 200 simulation runs, thirty statistically independent simulations will be run, each one with a different seed for the random number generator, in order to allow the computation of 95% confidence intervals for the obtained sample mean [OpNet 2004a].

This chapter established the simulation design. The next chapter will attempt to validate the model with a test case, and provide evidence of fairness of each MAC protocol considered.

V. MODEL VALIDATION

A. INTRODUCTION

The model validation is done with a simple linear topology, which contains a single data source to allow comparison of the simulation result with the analytical computation. Following this, specific simulation results will be used to evaluate the fairness of the study, especially regarding the CA/MS MAC mechanism. We conducted post-simulation runs in order to understand the unexpected results obtained from the execution of the simulation design defined in Chapter IV. The unexpected results presented in details in Chapters VI and VII, indicate a superior performance of the Aloha-like MAC mechanism, when compared with the CA/MS MAC scheme.

B. SIMPLE LINEAR TOPOLOGY

The topology in Figure 11, where the data originated in a single source (the sensor), travels in a hop-by-hop fashion to the gateway. The nodes are 1200 m apart from each other.

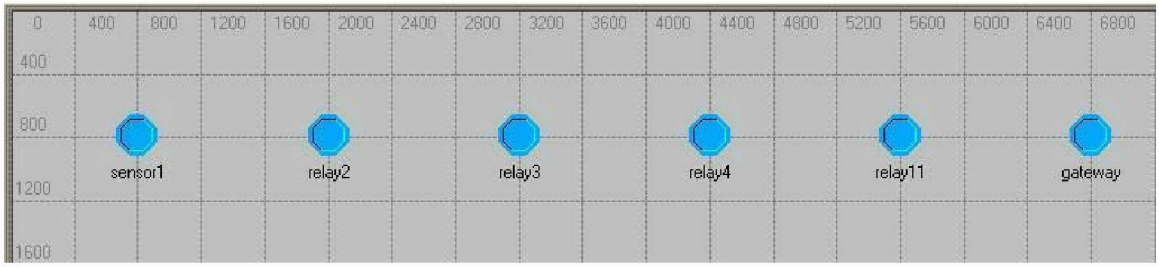

Figure 11. Linear topology with a single source

Four different simulations were run with the following settings:

- Data Frame Payload Size (DFPS): 1024 | 512 bits.
- Packet Generation:
  - Packet Size: 128 bytes = 1024 bits.
  - Inter-arrival Time: constant (80).
- Network Mode: CA | Aloha-like
- Simulation Time: 1 hour.
- Seed: 128

The results are presented in the following graph:

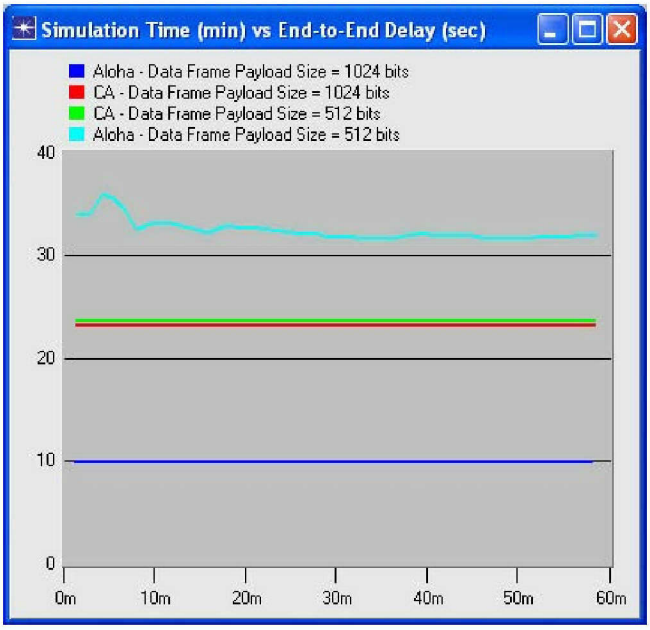

Figure 12. Simulation Time vs End-to-End Delay for a linear topology

As expected, the two simulation runs with the CA/MS and the Aloha-like mechanism with one frame per packet (DFPS equal to 1024 bits) show a constant delay throughout the simulation. This behavior was due to the low load introduced (i.e., one packet every 80 s) that allowed the arrival of the packet to its final destination before a new one was generated. On the other hand, the end-to-end delay in Aloha-like mechanism is not constant when there are two frames per packet (DFPS equal to 512 bits). This is caused by the random element in the back-off: after sending each frame, the node performs a back-off for a random number of slots. Note that the Aloha-like MAC protocol may be enhanced with a modification where it only performs a back-off when a timeout for an acknowledgment occurs. The current implementation makes the node perform a back-off after a successfully transmitted frame (ACK received), regardless of whether the node has more frames to transmit. This subject will be expanded in Chapter VIII in recommendations for future work.

The constant delays showed in Figure 12 are as follows:

- Aloha-like $_{\text{with DFPS = 1024 bits}} = 9.92$ s
- CA $_{\text{with DFPS = 1024 bits}} = 23.1$ s

- $CA_{\text{with DFPS = 512 bits}} = 23.55$ s

In the next section, the simulation results will be validated with the analytical calculation of the end-to-end delay in a linear topology.

C. ANALYTICAL CALCULATIONS FOR THE END-TO-END DELAY IN A LINEAR TOPOLOGY

For the next sub-sections, consider the following definitions:

- t – transmission delay.
- p – propagation delay.
- The subscript *f* relates to a data frame. For example, t_f represents the transmission delay for a data frame.
- The subscript *ack* relates to an acknowledgement.
- The subscript *rts* relates to a RTS.
- The subscript *cts* relates to a CTS.

1. The Aloha-like MAC Protocol

In the case of the Aloha-like MAC protocol, the end-to-end delay between nodes is illustrated in Figure 13.

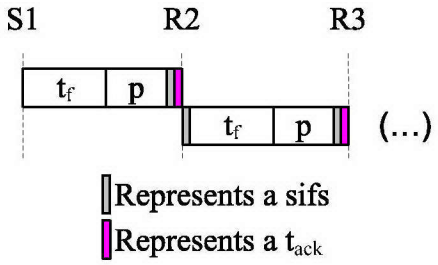

Figure 13. Partial pictorial representation of the delay between nodes for the Aloha-like protocol

The Node S(ensor)1 sends the frame immediately, incurring in a transmission delay (t_f) followed by a propagation delay (p). In linear networks such as this one, this occurs the number of times that is equal to the number of hops (#hops). Under such conditions, for each node that acts as a receiver / forwarder, then it incurs a SIFS delay before the ACK, a transmission delay for the ACK, and a SIFS delay before forwarding the data frame to the next node in line. In the network, this happens four times (the number of relay nodes - #relay), because this is not applicable for Node S1, and in the Gateway, the data frame is considered received as soon as the node ends its reception.

Therefore, the end-to-end delay can be expressed as:

$$\begin{aligned} ete\ delay_{Aloha-like-1F} &= \#hops\left(t_f + p\right) + \#relay\left(sifs + t_{ack} + sifs\right) \\ &= 5\times\left(1.104 + 0.8\right) + 4\times\left(0.01 + 0.08 + 0.01\right) \\ &= 9.52 + 0.40 \\ &= 9.92\,s \end{aligned}$$

with

$$t_f = \frac{data\ frame\ size}{data\ rate} = \frac{header + payload}{data\ rate} = \frac{80 + 1024}{1000} = 1.104\,s$$

$$t_{ack} = \frac{ack\ frame\ size}{data\ rate} = \frac{80}{1000} = 0.08\,s$$

$$p = \frac{distance}{propagation\ speed} = \frac{1200}{1500} = 0.8\,s$$

$$sifs = 0.01\,s$$

This result validates the one obtained with the simulation for the Aloha-like MAC protocol with one frame per packet (run reported in Section B).

2. The CA MAC Protocol

In the case of the CA/MS MAC scheme, there are additional delays. Figure 14 illustrates the delays incurred between nodes.

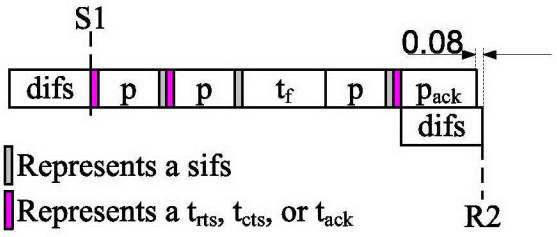

Figure 14. Partial pictorial representation of the delay between nodes for the CA/MS MAC protocol

Because of the low load, the first DIFS at Node S1 is absorbed during the time the receiver is idle. Until Node R(elay)2 receives the data frame, the process incurs in the following delays: RTS transmission delay, CTS transmission delay, data frame transmission delay, three propagation delays, and two SIFS delays. This process occurs a number of times that is equal to the number of hops. Then, as a receiver / forwarder node, additional delays need to be computed before the node initiates the RTS / CTS exchange

with the next node in line: a SIFS delay before transmitting the ACK frame, the ACK transmission delay, the ACK propagation delay, and a small portion of the DIFS that was not absorbed by the time that the receiver was idle. The ACK propagation delay, because the NAV reservation must be accounted for, is calculated with the maximum reachable distance of the node (1500 m).

Therefore, the end-to-end delay can be expressed as:

$$
\begin{aligned}
ete\ delay_{CA-1F} &= \#hops \times \left(t_{rts} + t_{cts} + t_f + 3p + 2sifs\right) \\
&\quad + \#relay \times \left(sifs + t_{ack} + p_{ack} + \left(remaining\ difs\right)\right) \\
&= 5 \times \left(0.08 + 0.08 + 1.104 + 3 \times 0.8 + 2 \times 0.01\right) + 4 \times \left(0.01 + 0.08 + 1 + 0.08\right) \\
&= 18.42 + 4.68 \\
&= 23.1s \\
&with \\
t_f &= \frac{frame\ size}{data\ rate} = \frac{header + payload}{data\ rate} = \frac{80 + 1024}{1000} = 1.104s \\
t_{rts} &= t_{cts} = t_{ack} = \frac{frame\ size}{data\ rate} = \frac{80}{1000} = 0.08s \\
p_{ack} &= \frac{reachable\ distance}{propagation\ speed} = \frac{1500}{1500} = 1s
\end{aligned}
$$

This result *validates* the one obtained with the simulation run (reported in Section B) for the CA/MS MAC protocol with one frame per packet.

Note that the performance of the CA/MS MAC protocol may be improved with a small change in the behavior, replacing the explicit ACK by an implicit ACK in the RTS to the next node in line. Although this behavior was not implemented in the current model, one may have fairly good idea about the impact of that change when computing what the end-to-end delay would be with that change:

$$
\begin{aligned}
ete\ delay_{CA-1F-Enhanced} &= \#hops \times \left(t_{rts} + t_{cts} + t_f + 3p + 2sifs\right) + \#relay \times \left(difs\right) \\
&= 5 \times \left(0.08 + 0.08 + 1.104 + 3 \times 0.8 + 2 \times 0.01\right) + 4 \times \left(1.08\right) \\
&= 18.42 + 4.32 \\
&= 22.74s
\end{aligned}
$$

The calculation does not show a significant improvement. But if one also reduces the DIFS period to zero, the end-to-end delay would be 18.42 s, still higher than the end-to-end delay showed in the case of the Aloha-like MAC protocol (almost double). The possibility of enhancing the CA MAC protocol will be explained further in Chapter VIII in the recommendations for future work.

Finally, in the case of the CA/MS MAC protocol with two frames per packet, one needs to add to the end-to-end delay calculated for the case of one frame per packet, one SIFS duration and the additional data frame header transmission delay per sending node, yielding:

$$\begin{aligned} ete\ delay_{CA-2F} &= ete\ delay_{CA-1F} + \#hops\left(sifs + \frac{data\ frame\ header\ size}{data\ rate}\right) \\ &= 23.1 + 5 \times \left(0.01 + \frac{80}{1000}\right) \\ &= 23.55s \end{aligned}$$

This result *validates* the one obtained with the simulation run (reported in Section B) for the CA/MS MAC protocol with two frames per packet.

D. VERIFICATION OF FAIRNESS

Another concern regarding the behavior of model is whether it allows the different nodes that are generating traffic to transmit their packets in a fair manner. In order to illustrate that, two graphs are shown, with the number of back-off slots performed by comparable nodes for the two different network topologies: the tree and the grid topology. The following graphs summarize the results of simulation runs that will be presented in the next two chapters.

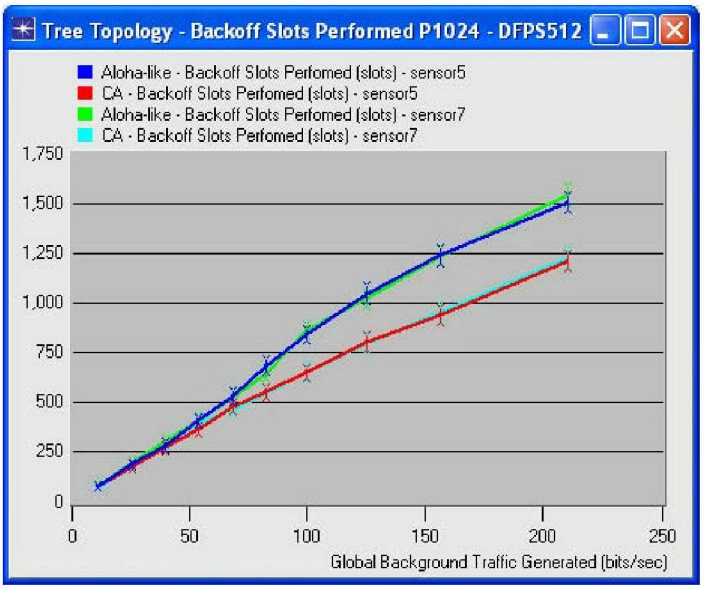

Figure 15. Load vs Backoff slots of Sensor5 and Sensor7 in Figure 21, with different MAC protocol, in a tree topology with a packet size of 1024 bits and a DFPS of 512 bits

The number of backoff slots is an indication of the fairness of how the MAC scheme treats each node. The nodes plotted in Figure 15 and 16 were chosen because they serve the same traffic pattern, and the node to which they are sending data serve a comparable traffic pattern, as well. If these two conditions were not fulfilled, the number of back-off slots performed would be different, because of the different load conditions under which the nodes are. Thus, when evaluating the number of backoff slots of two comparable nodes, as depicted, the curves for the same MAC scheme are statistically equal, with a confidence interval of 95%. Therefore, for nodes in comparable positions, both schemes are giving them the same opportunity to transmit their data.

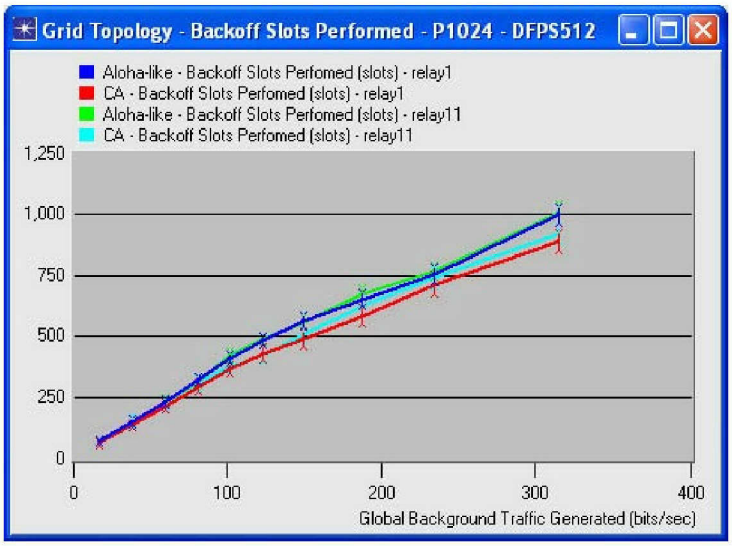

Figure 16. Backoff slots of Relay1 and Relay11 in Figure 33, with different MAC protocol, in a grid topology with packet size of 1024 bits, and a DFPS of 512 bits.

E. POST-SIMULATION EXPERIMENTS

The results presented in the next two chapters show an unexpected performance advantage of the Aloha-like MAC protocol over the CA/MS scheme in the case where the packet size is 1024 bits and contain one frame. This is surprising, since it is commonly accepted that in wireless radio-based networks the CA scheme outperforms the Aloha approach. In order to rule out the possibility that the implementation used here may have unfairly penalized the CA/MS scheme, and in order to get some insight into other reasons for these unexpected results, the decision was made to run extra simulations. With these additional simulation runs, the experimenter's intent was to make a two-fold verification: firstly, to be sure that the CA/MS MAC scheme was not penalized, forcing the nodes sensing the medium free for too much time before they are allowed to transmit (DIFS' default value is 1.08 s). We accomplish this setting the DIFS equal to zero); secondly, to have an idea of how the propagation speed was affecting the relative performance of the protocols considered.

1. Simulation Results with DIFS Equal to Zero

Six-hundred simulations were run with the grid topology and the following settings:

- DIFS = 0;

- Data Frame Payload Size (DFPS): 1024.
- Packet Generation:
  - Packet Size: 128 bytes = 1024 bits.
  - Inter-arrival Time: constant distribution with 10 means: 48 | 65 | 80 | 102 | 120 | 144 | 180 | 240 | 360 | 720.
- Network Mode: CA | Aloha-like
- Simulation Time: 1 hour.
- Seed: 30 different seeds from 128 to 157.

The results are presented in the following graphs:

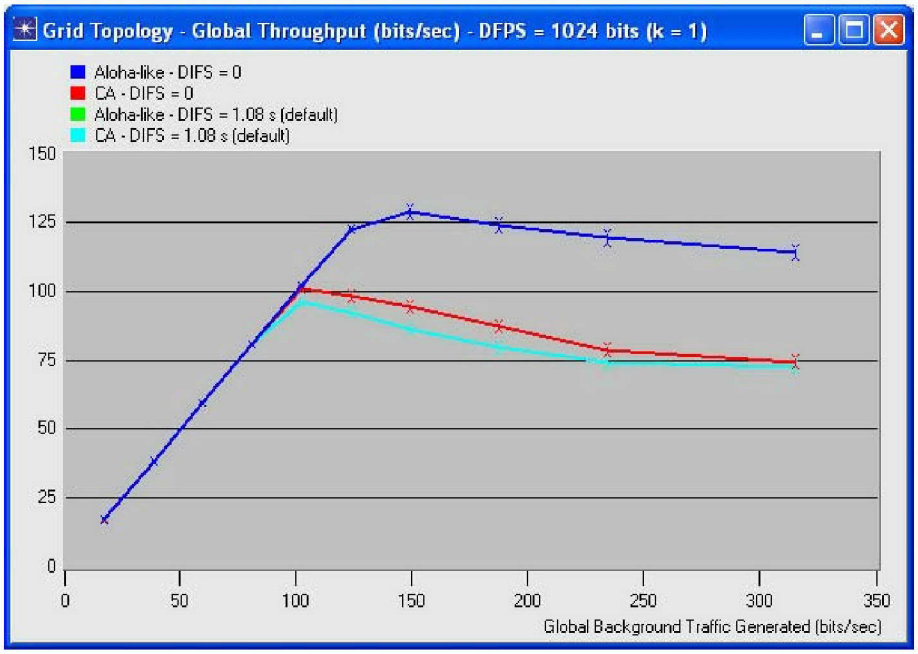

Figure 17. Load vs Throughput with DIFS = 0 | 1.08 in a grid topology

The throughput for the Aloha-like mechanism does not change when different DIFS values were used, as expected, because DIFS is not a considered parameter in this MAC scheme. In the case of the CA/MS protocol, the value of DIFS has an impact on performance. With DIFS equal to zero, the performance increases only a little. This was as expected, because from the analytical calculations made in Section C.2. for a linear topology, the DIFS duration overlaps the time to propagate the ACK. Therefore, a dramatic improvement in performance was not anticipated.

Figure 18 presents the same type of graph, but now for the end-to-end delay. The observations are the same. The performance of the Aloha-like scheme does not change with different DIFS values, and the performance of the CA/MS mechanism improves only a little bit when DIFS equal to zero is used.

Figure 19 presents a graph with a zoom to the left lower corner of the graph presented in Figure 18, where the differences in performance between the CA/MS scheme with the two different DIFS, and the Aloha-like scheme are highlighted.

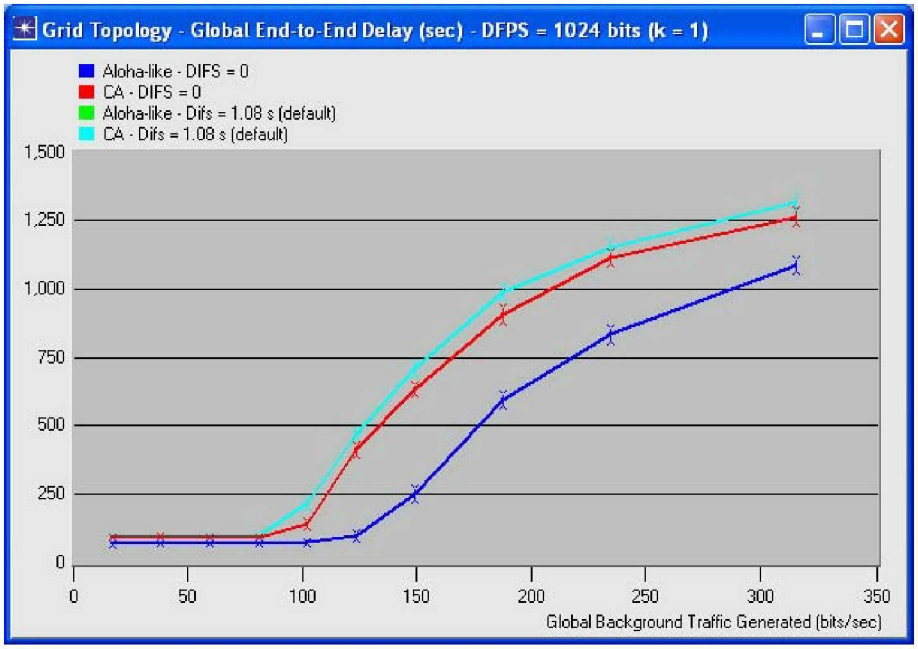

Figure 18. Load vs End-to-End with DIFS = 0 | 1.08 in a grid topology

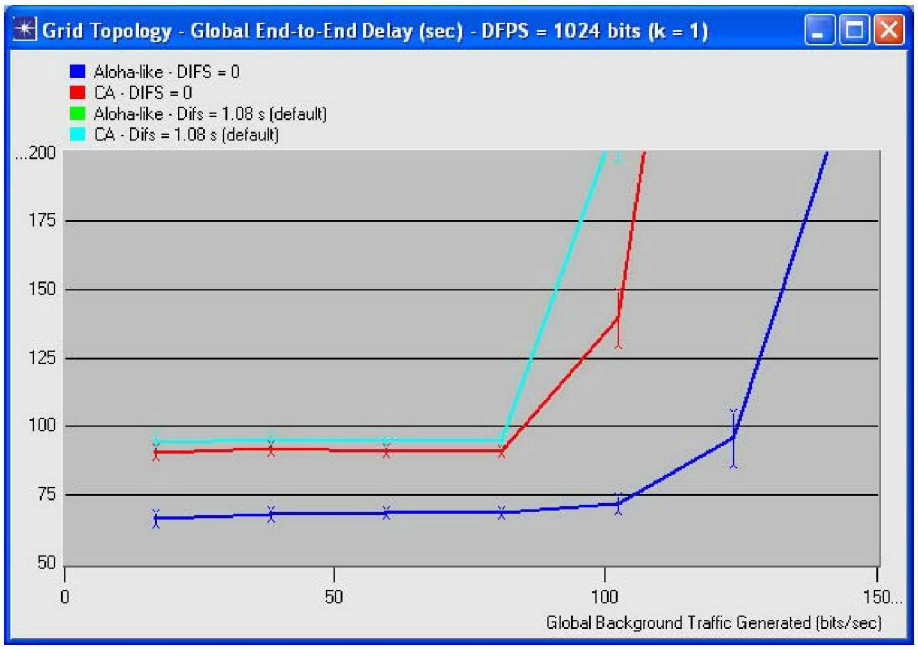

Figure 19. Zoom to the left lower corner of the graph presented in Figure 18

2. Change in Performance with the Propagation Speed

This experiment set up a simple simulation, looking for the influence of the propagation speed on the relative performance of the two MAC mechanisms considered. One simulation was run with the following setup:

- Data Frame Payload Size (DFPS): 1024.
- Packet Generation:
  - Packet Size: 128 bytes = 1024 bits.
  - Inter-arrival Time: constant with mean 65.
- Network Mode: CA | Aloha-like
- Simulation Time: 1 hour.
- Seed: 128
- Propagation Speed: 8 levels from 1500 m/s to 5000 with steps of 500.

The results are presented in Figure 20 below:

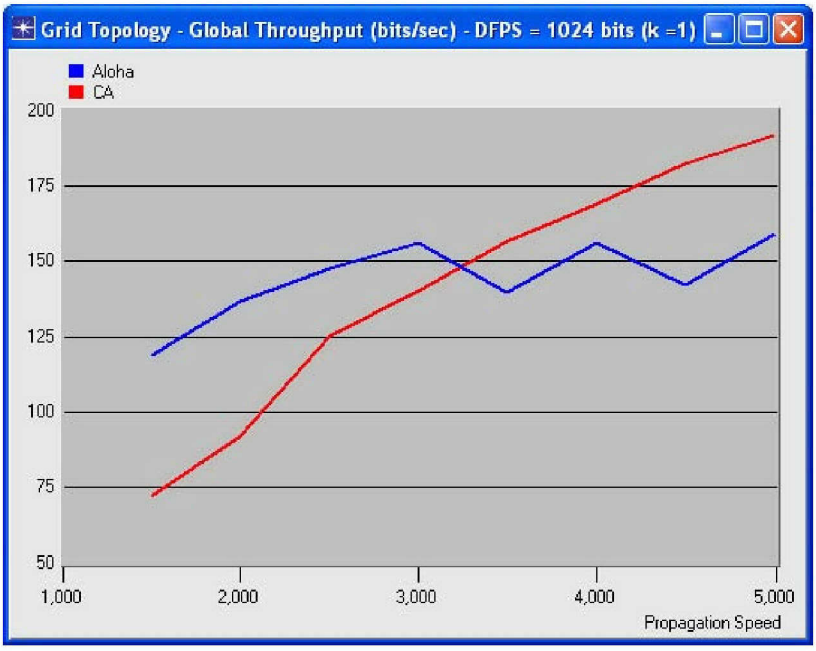

Figure 20. Propagation Speed vs Throughput with a DFPS of 1024 bits in a grid topology

The graph in Figure 20 clearly shows a relation between the performance of the CA/MS MAC protocol and the propagation speed of the medium. This result is not statistically valid, however, because only one simulation run was made. The trend, however, is quite strong.

The throughput of the Aloha-like MAC mechanism is not sensitive to the change of the propagation speed of the medium. Since this is an uncoordinated mechanism, the increase on the propagation speed increases the speed of all the traveling frames. With everything being the same, the amount of collisions and other performance penalties may not change, and the overall performance seems to maintain an almost constant trend.

The initial poorer performance of the CA/MS scheme may be related to the cost of the three-fold propagation delay that the RTS-CTS handshake incurs, as well as the effective utilization of the channel for data transmission. It seems that only after some significant increase of the propagation speed of the medium, the handshake overhead becomes small enough to allow the CA/MS scheme to outperform the Aloha-like mechanism.

Although Figure 20 suggests a possible explanation for the performance reported in the following two chapters, this relation was not studied in the present thesis in the depth necessary to draw principles that might apply more generally.

Chapter VI will present the results of the simulation runs for the tree topology network. Chapter VII will then present the results for the grid topology network.

THIS PAGE INTENTIONALLY LEFT BLANK

VI. SIMULATION RESULTS – TREE TOPOLOGY

A. NETWORK SETUP

In the tree topology (Figure 21), all the sensor nodes generate background and non-periodic traffic. The relay nodes do not generate traffic. Their sole function is to forward the message to the next relay node toward the gateway. Therefore, relay nodes can receive traffic from the previous relay node in the path, and from all the sensor nodes in their 1-hop neighborhood.

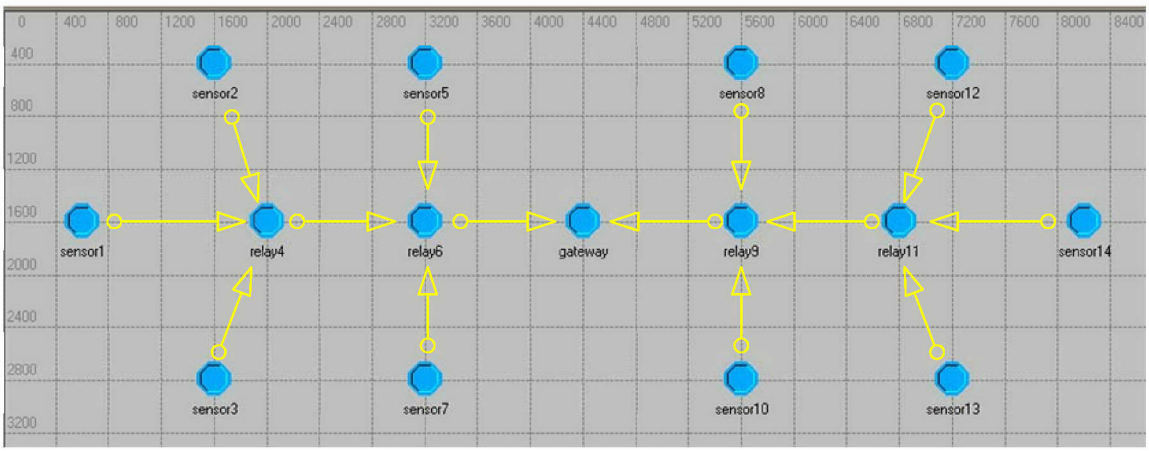

Figure 21. Tree Topology Network Layout

Figure 21 was extracted from the OnNet Project Editor at the network level. The arrows were added to the picture only for illustration purposes. They illustrate both the data flows and the fixed routing path. The nodes are placed at such a distance that the signal of each sensor node is only able to reach its relay node. Note that in this topology one has the typical hidden terminal problem of wireless networks. For example, if Sensor7 wants to send data to its relay node, Relay6, it needs to compete for the media with Sensor5 and Relay4, neither of which Relay6 can directly sense. Relay6 performs that mediation and, if it receives non-colliding requests, will issue permission to the first one received.

B. BACKGROUND TRAFFIC PERFORMANCE

The results presented in the following sub-sections were obtained with the following settings (600 simulation runs for each DFPS):

- Data Frame Payload Size (DFPS): 128 | 256 | 512 | 1024 bits.

- Packet Generation:
  - Packet Size: 128 bytes = 1024 bits.
  - Inter-arrival Time: constant distribution with 10 means: 48 | 65 | 80 | 102 | 120 | 144 | 180 | 240 | 360 | 720.
- Network Mode: CA/MS | Aloha-like
- Simulation Time: 1 hour.
- Seed: 30 different seeds from 128 to 157.

1. Throughput

Figure 22 and 23 present the results of background load versus global throughput when different frame sizes were used. The tree topology was used in this simulation. The results show, as expected, that the CA/MS MAC scheme is not greatly affected by the change in number of frames per packet because the nodes send the entire data packet after one RTS/CTS exchange, regardless of the number of frames per packet. The differences between the performance results of the CA/MS scheme that were obtained are due only to the SIFS duration and data frame header transmission delay added for each additional frame per packet (see Chapter V – Section C.2. for additional details).

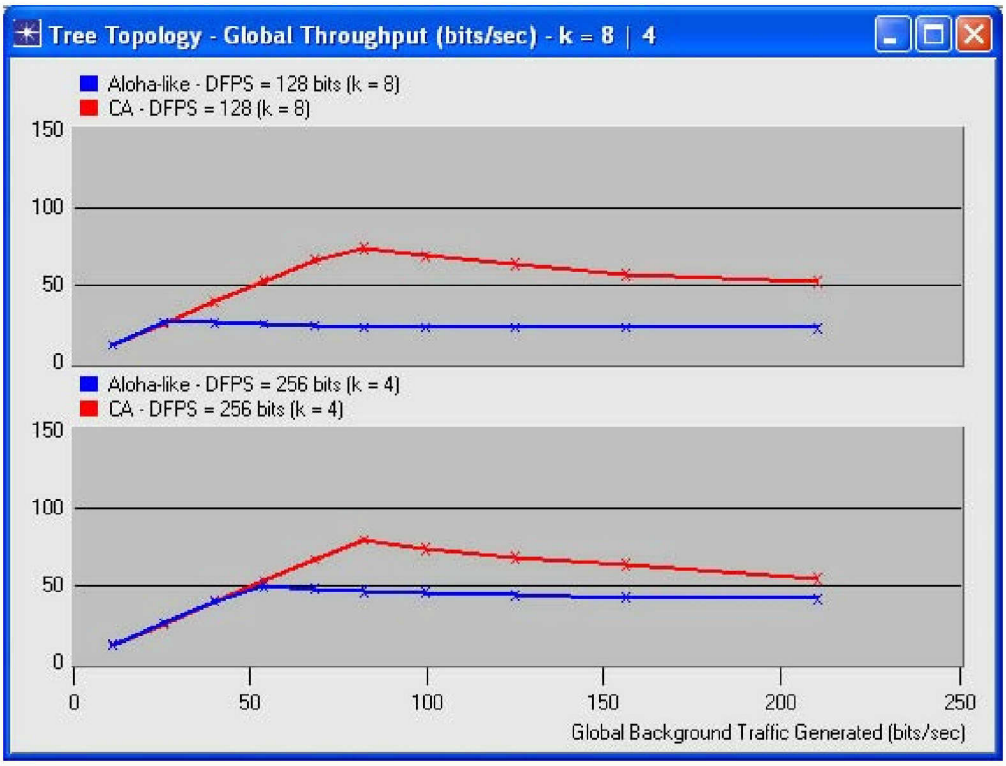

Figure 22. Tree Topology – Load vs Throughtput – DFPS = 128 | 256 bits

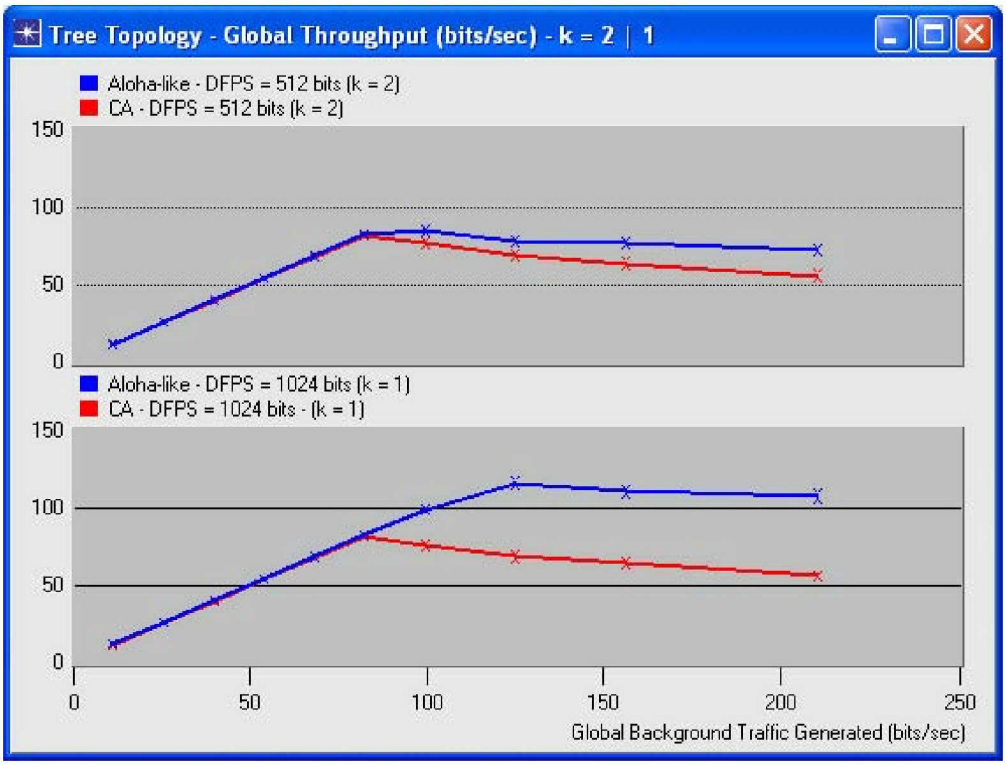

Figure 23. Tree Topology – Load vs Throughput – DFPS = 512 | 1024 bits

On the other hand, the Aloha-like MAC mechanism is greatly influenced by how fragmented the packet is. That is explained by the need to backoff after a successful frame transmission, as was implemented in this model. Considering the delay bandwidth product for these implementation settings, the Aloha-like MAC scheme is only able to "fill the pipe" with a DFPS equal to the packet size. On the other hand, the CA/MS MAC scheme, in all situations, i.e., when using different number of frames per packet, is able to "fill the pipe". Unexpectedly, the Aloha-like, when decreasing the number of frames (k) per packet, starts outperforming the CA/MS MAC scheme with two frames per packet, and, clearly, has a much better performance with a number of frames per packet equal to one. The explanation for these results is related with the variation on the relative performance of both protocols when the propagation speed is changed, as described in Chapter V.

2. End-to-End Delay

The next graphs (Figures 24 and 25) show the performance in terms of end-to-end delay of the two MAC schemes with different DFPS.

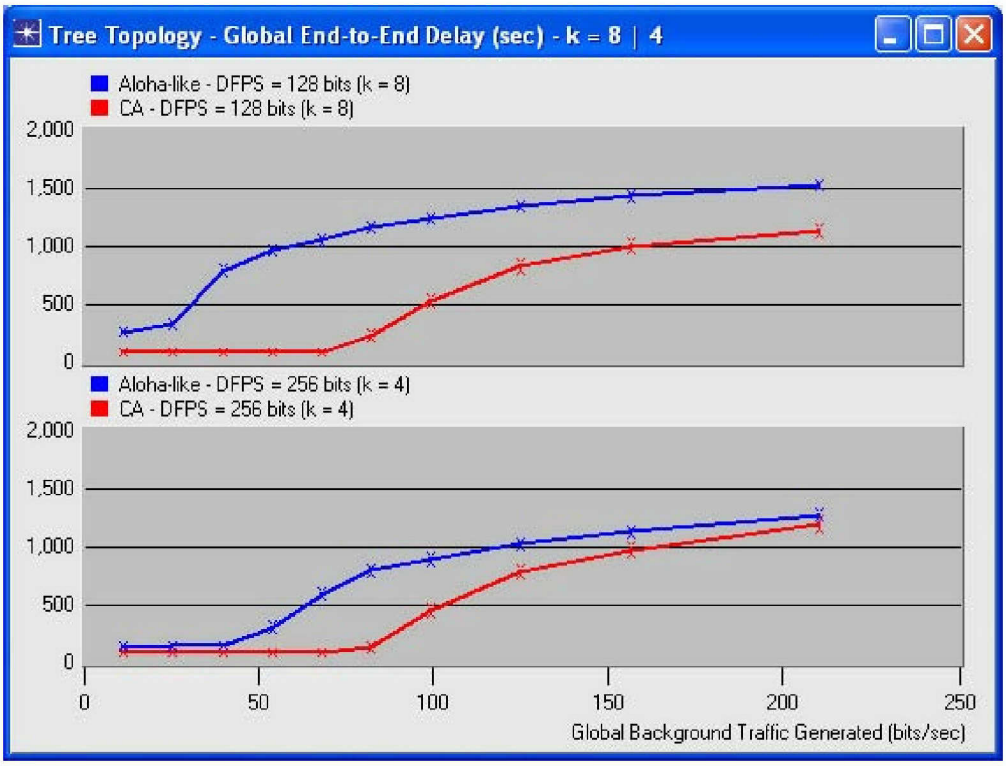

Figure 24. Tree Topology - Load vs End-to-End Delay – DFPS = 128 | 256 bits

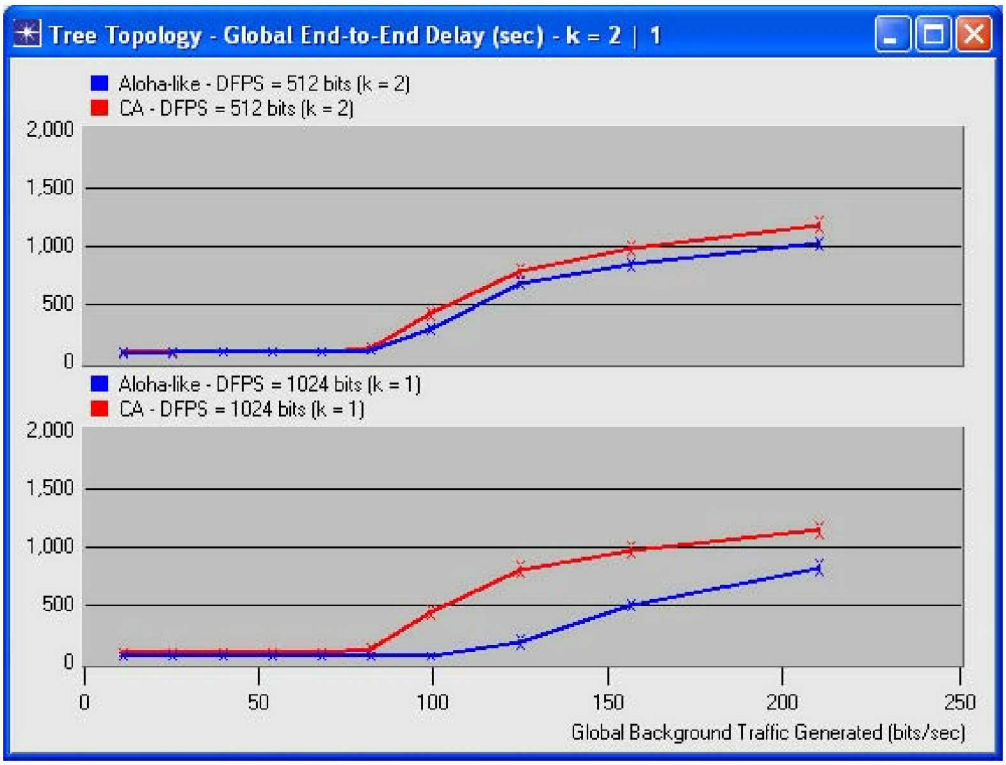

Figure 25. Tree Topology - Load vs End-to-End Delay – DFPS = 512 | 1024 bits

The end-to-end delay metrics reinforces the observation that the Aloha-like scheme performance improves when the number of frames per packet decreases. The CA/MS MAC scheme, as expected, does not change noticeably when the number of frames per packet is changed. Figure 26 shows a zoom into the left lower corner of the graphs plotted in Figure 25 to demonstrate the huge difference in the performance of the two MAC schemes.

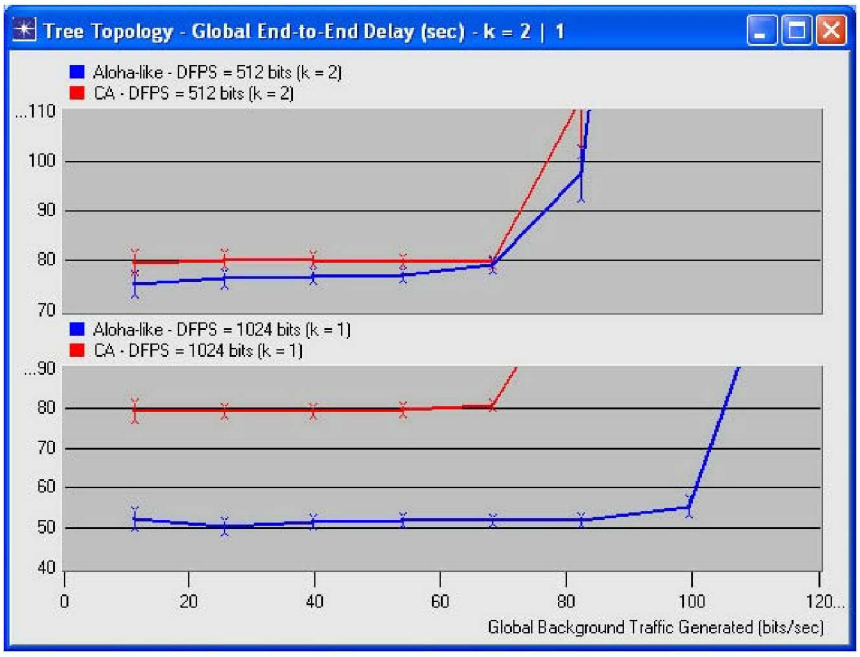

Figure 26. Zoom into the lower left corner of the graphs plotted in Figure 25

3. Collision Events and Packets in Queue

In order to provide some insight into this unexpected result, the total collision events, as well as packets in queue at the end of simulation, was investigated. The collisions graph (Figure 27) shows that, in the case of CA/MS there is not a noticeable difference in the number of collisions when the number of frames per packet is changed. Again, this is related with the behavior of sending the entire packet under the same RTS-CTS exchange, regardless of the number of frames per packet. On the other hand, in the case of Aloha-like, the change in significant, decreasing when the number of frames per packet also decreases. The decreasing number of transmitted frames seems to be the relevant factor affecting performance. Although the data transmitted is the same, the fact that they are transmitted in the same "transmitting window" appears to decrease the number of collisions and improves the performance dramatically.

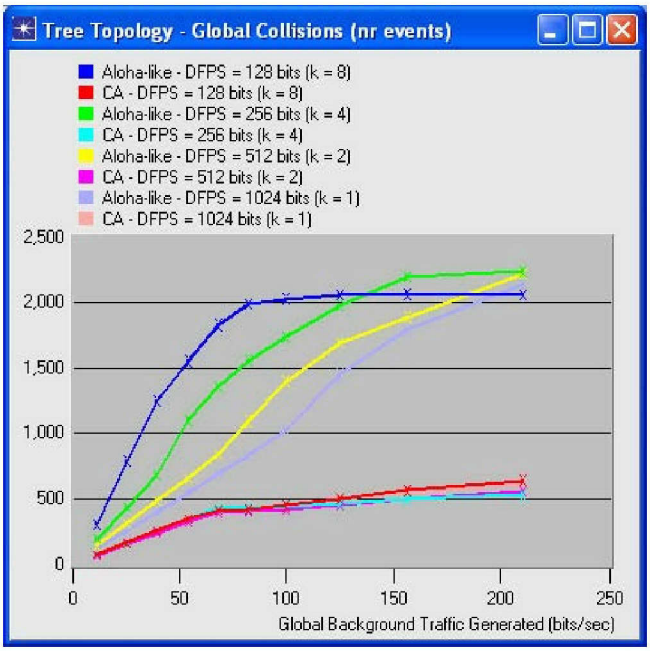

Figure 27. Tree topology – Load vs Collisions for the two MAC schemes and with different number of frames per packet

The next graph shows the number of packets in the queues of all nodes when the simulation terminates.

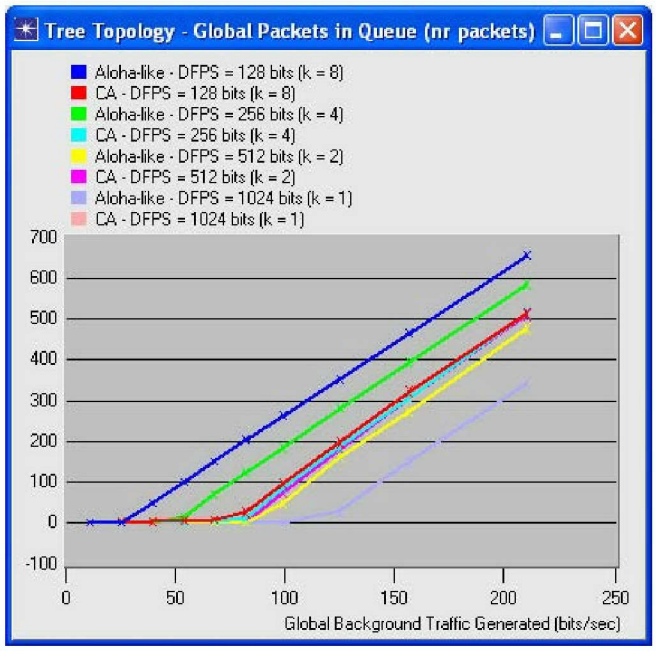

Figure 28. Tree Topology – Load vs Packets in Queue for the two MAC schemes and with different number of frames per packet

The graph presented in Figure 28, shows that the different settings plotted are able to handle the traffic generated with low network loads. However, when the load in the network is increased, all of them reached a point beyond which they are unable to process all the requests, represented in the data by the increased number of packets in queue at the end of simulation. In the case of the CA/MS MAC scheme, and considering the plotted curves for different frame sizes, there is not much variation in the number of packets in queue when the simulation terminates. In the case of the Aloha-like MAC mechanism, it shows both, the worst (k = 8) and the best performance (k = 1), in accordance with the behavior already reported, where this scheme increases its performance when the number of frame per packet is decreased.

C. NON-PERIODIC TRAFFIC PERFORMANCE

The results presented in the following sub-sections were obtained with the following settings (600 simulation runs):

- Data Frame Payload Size (DFPS): 1024 bits.
- Background Traffic Packet Generation:
  - Packet Size: 128 bytes = 1024 bits.
  - Inter-arrival Time: constant distribution with 10 means: 48 | 65 | 80 | 102 | 120 | 144 | 180 | 240 | 360 | 720.
- Non-Periodic Traffic Packet Generation:
  - Packet Size: 640 bytes = 5120 bits.
  - Inter-arrival Time: exponential distribution arrival with a mean of 600 s.
- Network Mode: CA/MS | Aloha-like
- Simulation Time: 1 hour.
- Seed: 30 different seeds from 128 to 157.

It follows the presentation of some performance graphs of load vs throughput (Figure 29) and load vs end-to-end delay (Figure 30) to illustrate how the two MAC schemes perform relative to a non-periodic traffic pattern, introduced into the network over the periodic background traffic. Note that the load considered in the graphs is only the one introduced by the background traffic (Global Background Generated Traffic).

The throughput graph (Figure 29), shows that the throughput of both MAC schemes is affected when the load in the network increases. After a certain point, the

throughput of the non-periodic traffic served by the CA/MS scheme decreases at a greater rate than that served by the Aloha-like mechanism. The performance of each pair of the curves starts out following a similar pattern, with the CA/MS curve eventually showing a greater decrease in the throughput while the Aloha-like maintains its performance.

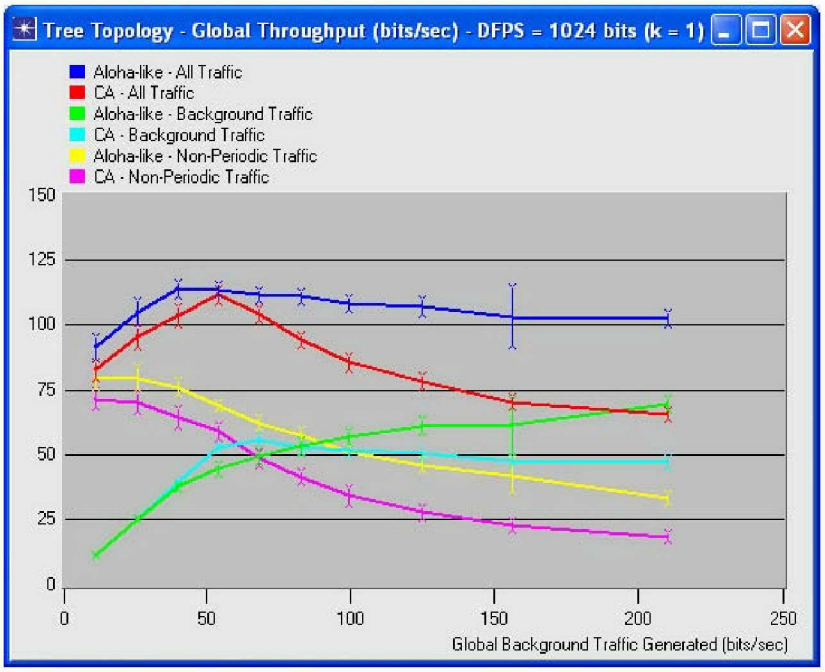

Figure 29. Tree Topology – Load vs Throughput of the different type of traffics. with a DFPS equal to 1024 bits

With the increase of network load, the throughput of the background traffic increases, and the throughput of the non-periodic traffic decreases.

Regarding the end-to-end delay (Figures 30, 31, and 32), the CA/MS mechanism shows a better performance in terms of non-periodic traffic performance. However, as reported earlier, the Aloha-like MAC scheme does not perform well with a traffic pattern of multiple frames per packet, such as in the used non-periodic traffic pattern.

Although in both cases the performance changes when the network load is increased, the CA/MS seems to be more sensitive to the increase in network load. Despite this phenomenon, considering the performance of only the non-periodic traffic, the CA/MS scheme has the best performance.

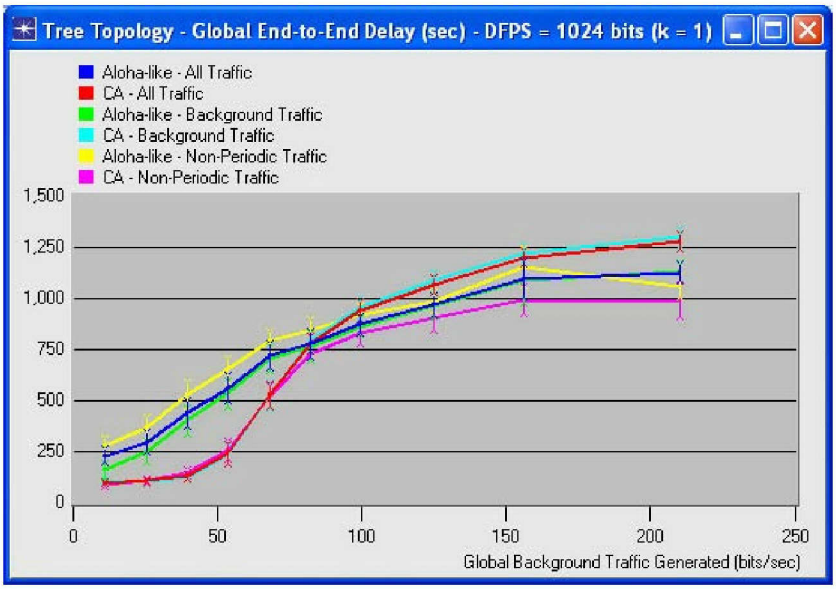

Figure 30. Tree Topology – Load vs End-to-End Delay of the different type of traffics, with a DFPS equal to 1024 bits

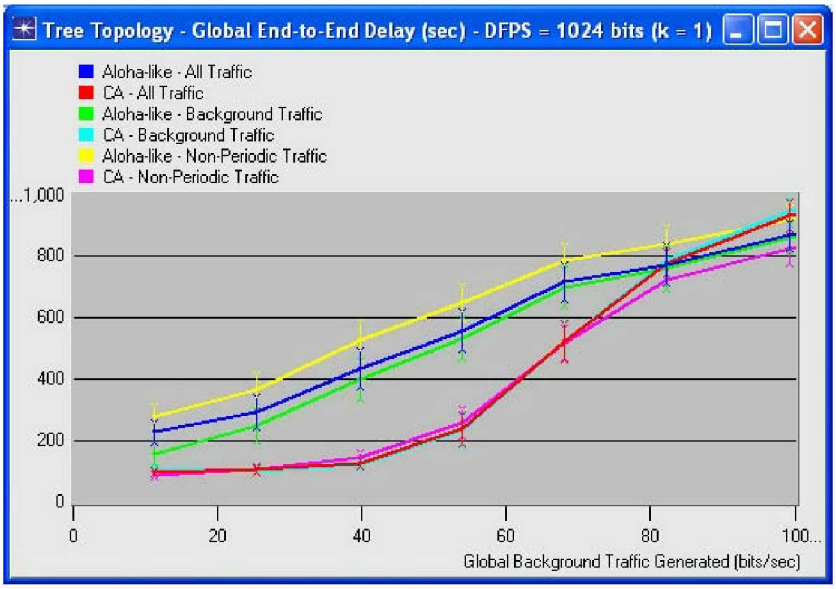

Figure 31. Zoom into the left lower corner of the graph on Figure 30

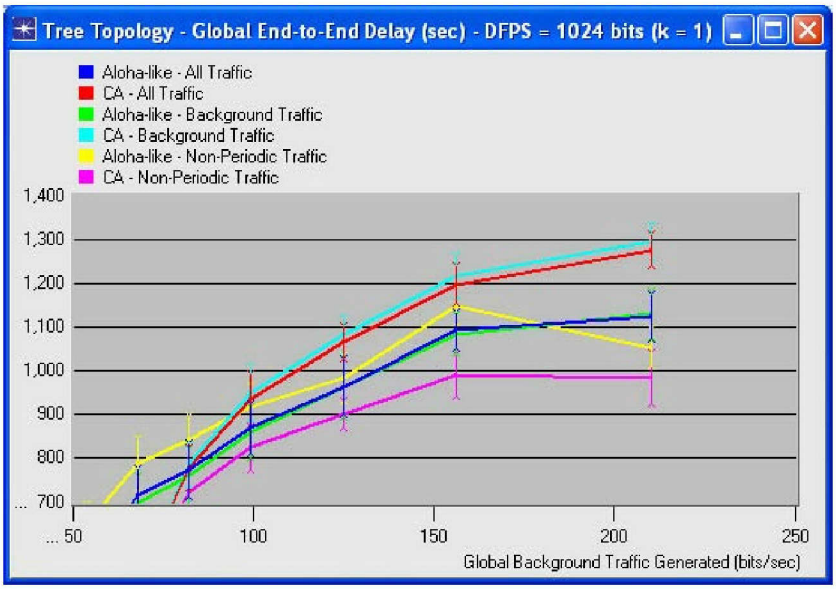

Figure 32. Zoom into the right higher corner of the graph on Figure 30

D. CHAPTER CONCLUSIONS

The results presented and observations made in this chapter can be summarized in the following conclusions:

- The performance of the CA/MS MAC schemes, both throughput and end-to-end delay, is marginally affected by the change in the number of frames per packet.
- The performance of the Aloha-like MAC scheme, both throughput and end-to-end delay, is greatly affected by the change in number of frames per packet.
- Although not fully explored, the Aloha-like MAC scheme appears to have an upper bound for throughput, closely related with the frame size.
- With one frame per packet, the Aloha-like MAC mechanism outperforms clearly the CA/MS MAC scheme in both, throughput and end-to-end delay. This is a result explained by the change in the relative performance of both schemes when the propagation speed of the medium is changed (see Chapter V for more details).
- Regarding the behavior of both schemes when the network is submitted to non-periodic traffic, the CA/MS MAC mechanism shows a better end-to-end delay performance. However, the nature of the non-periodic traffic introduces multiple frames packets, which we already demonstrated that it is a situation where the performance of Aloha-like MAC scheme is diminished.

The next chapter will present the simulation results for the grid topology.

THIS PAGE INTENTIONALLY LEFT BLANK

VII. SIMULATION RESULTS – GRID TOPOLOGY

A. NETWORK SETUP

The grid topology has only relay nodes (besides the gateway), all of which can generate internal traffic (Figure 33). This configuration attempts to represent a grid of monitoring sensors that also need to forward traffic from their neighbors.

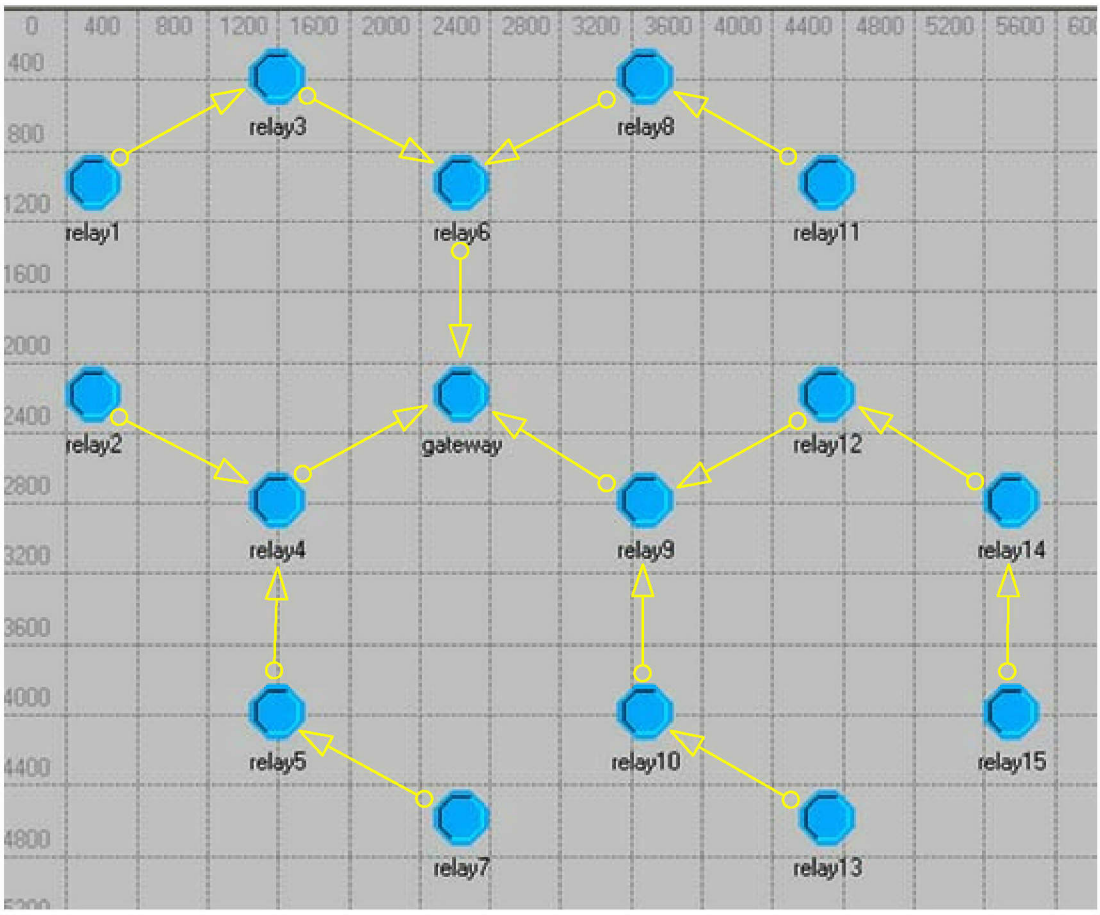

Figure 33. Grid Topology Network Layout

Figure 33 was extracted from the OnNet Project Editor at the network level. The arrows were added to the picture only for illustration purposes. They illustrate both the data flows and the fixed routing. The grid contains four hexagons, where nodes at adjacent vertices are 1200 m apart from each other. With this layout, each node will have a maximum of three reachable neighbors. For example, Relay9 can only reach the Gateway, Relay12 and Relay10. Relay9 cannot reach Relay4. It should be noted that,

despite the fixed routing and data flow, even if Node Relay13 is not forwarding its traffic through Node Relay15, or vice-versa, their signals ordinarily interfere with each other.

B. BACKGROUND TRAFFIC PERFORMANCE

The results presented in the following sub-sections were obtained with the following settings (600 simulation runs for each DFPS):

- Data Frame Payload Size (DFPS): 128 | 256 | 512 | 1024 bits.
- Packet Generation:
  - Packet Size: 128 bytes = 1024 bits.
  - Inter-arrival Time: constant distribution with 10 means: 48 | 65 | 80 | 102 | 120 | 144 | 180 | 240 | 360 | 720.
- Network Mode: CA/MS | Aloha-like
- Simulation Time: 1 hour.
- Seed: 30 different seeds from 128 to 157.

1. Throughput

Figures 34 and 35 plot the background load vs global throughput for the grid topology. Although there are some differences in exact values, the overall trend of relative performance is in line with what was observed for the tree topology network. The Aloha-like performs worse when there are several frames per packet, but it outperforms the CA scheme when there is one frame per packet. The CA/MS scheme shows insensitivity to the change of frames per packet. It seems that the results are prevalent across network topologies.

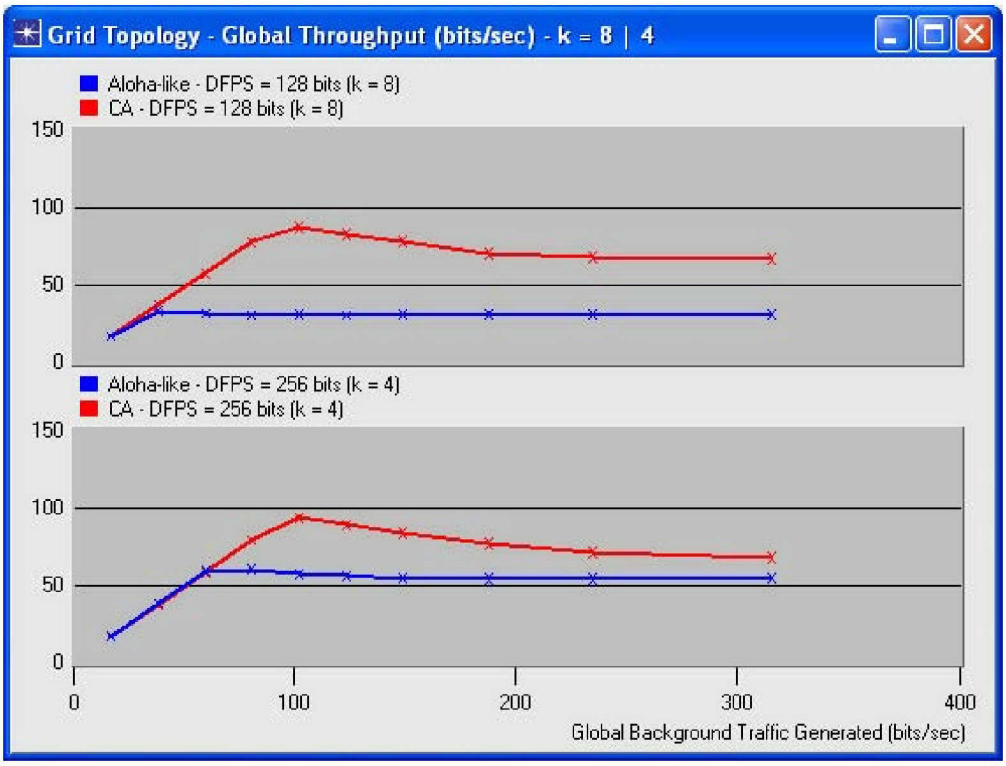

Figure 34. Grid Topology – Load vs Throughtput – DFPS = 128 | 256 bits

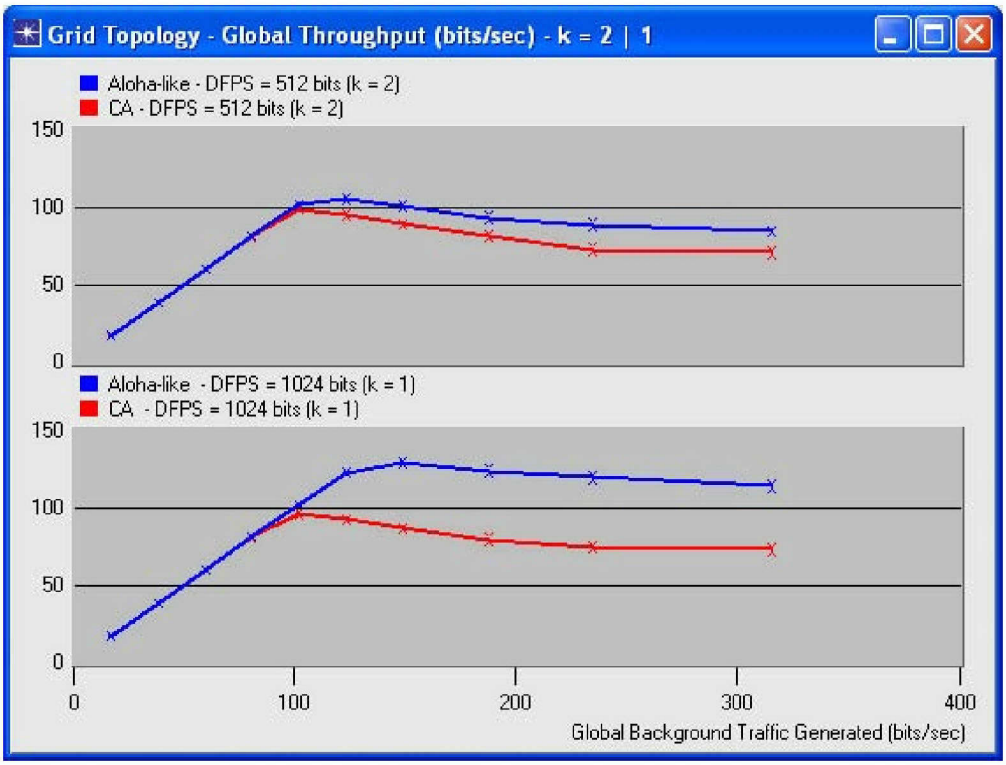

Figure 35. Grid Topology – Load vs Throughput – DFPS = 512 | 1024 bits

As discussed in Chapter VI, the more fragmented a packet is, the more back-off the Aloha-like MAC mechanism needs to perform, therefore affecting it overall performance. Additionally, the Aloha-like is able to "fill the pipe" when there is one frame per packet, whereas the CA/MS does that independently of the number of frames per packet. As was shown in the tree topology results, the Aloha-like MAC mechanism shows a better performance.

2. End-to-End Delay

The next graphs (Figures 36 and 37) show the performance in terms of end-to-end delay with the some settings, that is, comparison of the two MAC schemes with different DFPS.

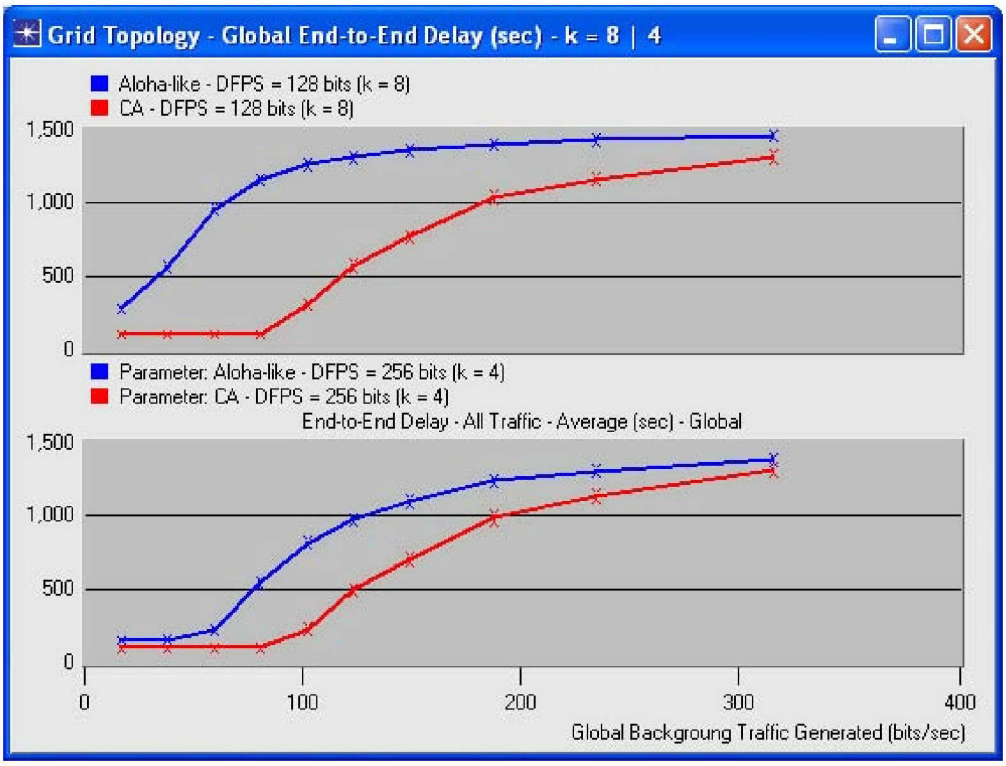

Figure 36. Grid Topology - Load vs End-to-End Delay – DFPS = 128 | 256 bits

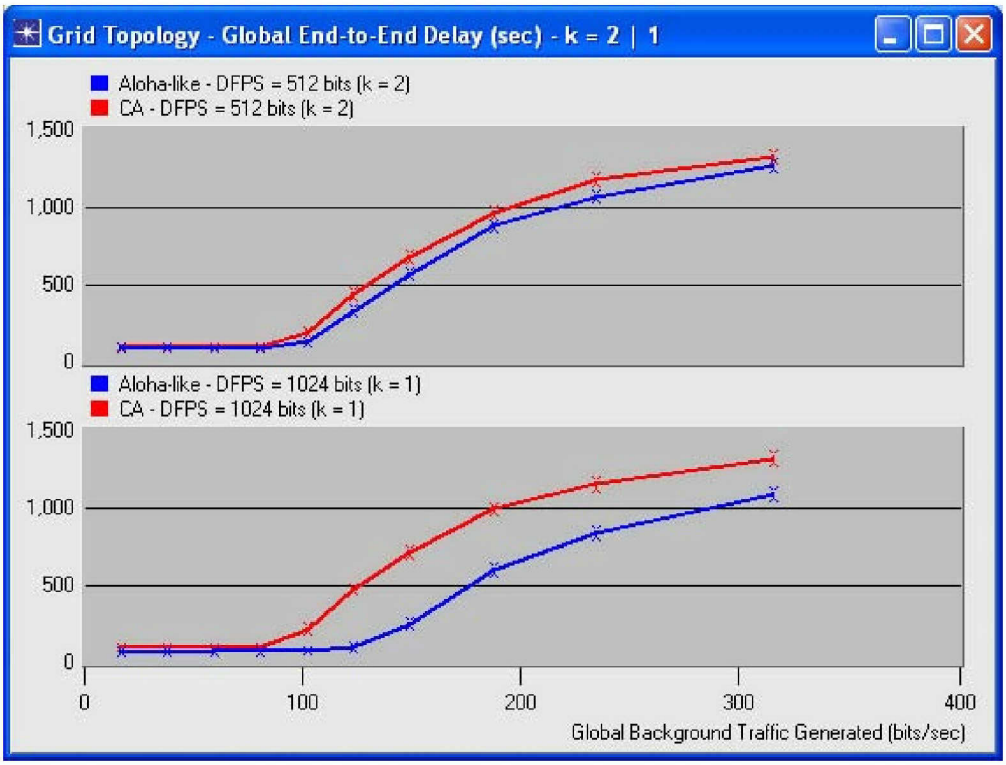

Figure 37. Grid Topology - Load vs End-to-End Delay – DFPS = 512 | 1024 bits

The end-to-end delay metrics in the grid topology shows similar results to the ones presented in the tree topology. It reinforces the insensitivity of the CA/MS MAC scheme to the change of frames per packet. Figure 38 shows a zoom into the left lower corner of the graphs plotted in Figure 37 to demonstrate the huge difference in the performance of the two MAC schemes.

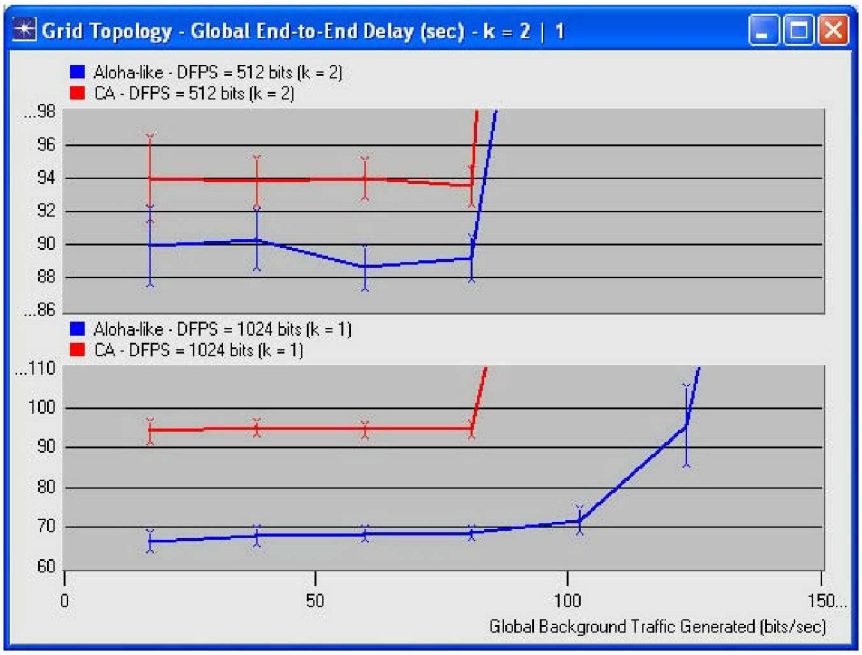

Figure 38. Zoom into the lower left corner of the graphs plotted in Figure 37

3. Collision Events and Packets in Queue

As in the tree topology presentation, the consolidation of collision events is shown, aggregating the packets in queue at the end of simulation.

The collisions graph (Figure 39) shows the same trend to the one presented for the tree topology. Again, this is related with the behavior of sending the entire packet under the same RTS-CTS exchange, regardless of the number of frames per packet. On the other hand, in the case of Aloha-like, the change is significant, decreasing when the number of frames per packet also decreases.

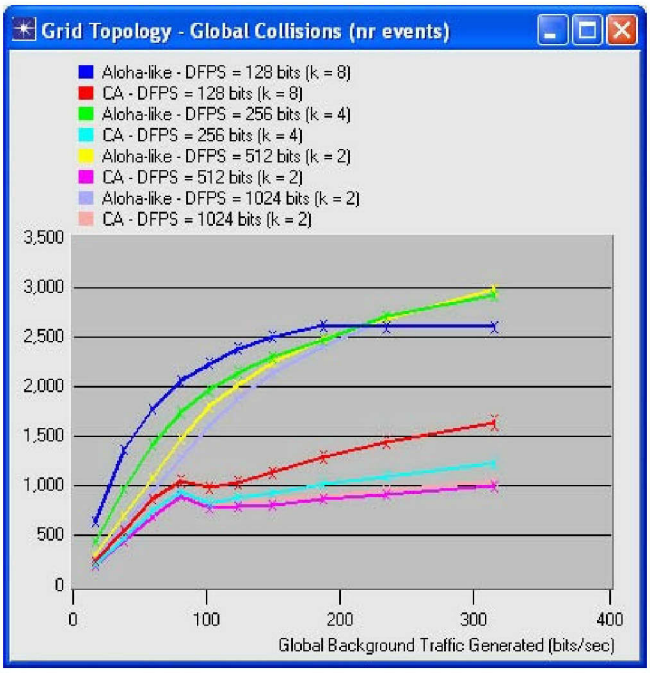

Figure 39. Grid topology – Load vs Collisions for the two MAC schemes and with different number of frames per packet

The next graph shows the number of packets in the queues of all nodes when the simulation terminates.

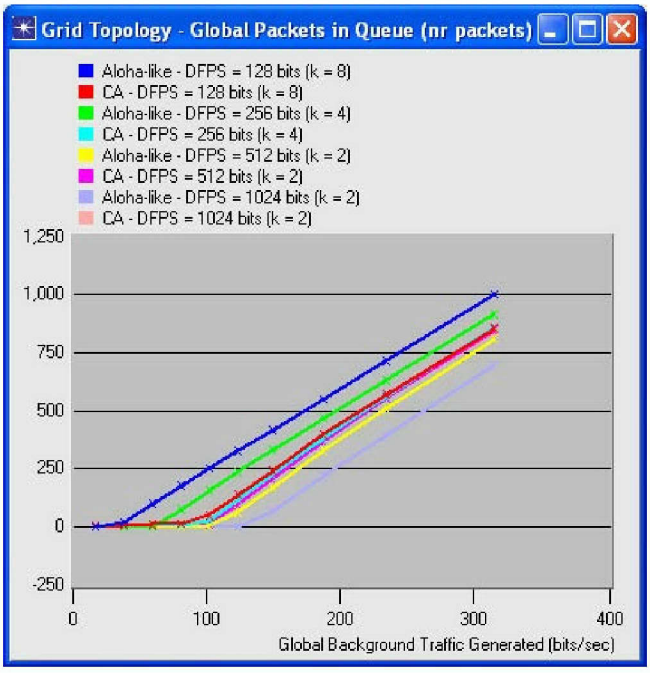

Figure 40. Grid Topology – Load vs Packets in Queue for the two MAC schemes and with different number of frames per packet

In the case of CA/MS, there is not much variation when we change the number of frames per packet. In the case of the Aloha-like, it starts with the worst performance, but when the number of frames per packet is decreased, it significantly outperforms the CA/MS scheme.

C. NON-PERIODIC TRAFFIC PERFORMANCE

The results presented in the following sub-sections were obtained with the following settings (600 simulation runs):

- Data Frame Payload Size (DFPS): 1024 bits.
- Background Traffic Packet Generation:
  - Packet Size: 128 bytes = 1024 bits.
  - Inter-arrival Time: constant distribution with 10 means: 48 | 65 | 80 | 102 | 120 | 144 | 180 | 240 | 360 | 720.
- Non-Periodic Traffic Packet Generation:
  - Packet Size: 640 bytes = 5120 bits.
  - Inter-arrival Time: exponential distribution arrival with a mean of 600 s.
- Network Mode: CA/MS | Aloha-like
- Simulation Time: 1 hour.
- Seed: 30 different seeds from 128 to 157.

It follows the presentation of some of the performance graphs of load vs throughput (Figure 41) and load vs end-to-end delay (Figure 42), to illustrate how the two MAC schemes perform relatively to a non-periodic traffic pattern introduced into the network over the background traffic. Note that the load considered in the graphs is only the one introduced by the background traffic (Global Background Generated Traffic).

The throughput graph (Figure 41), shows that the throughput of the non-periodic traffic in both MAC schemes is affected when the load in the network increases. The performance of the all traffic curve for both schemes follows, more or less, the same pattern. However, different than in the tree topology results, the CA/MS scheme outperforms Aloha-like when the load in the network is low.

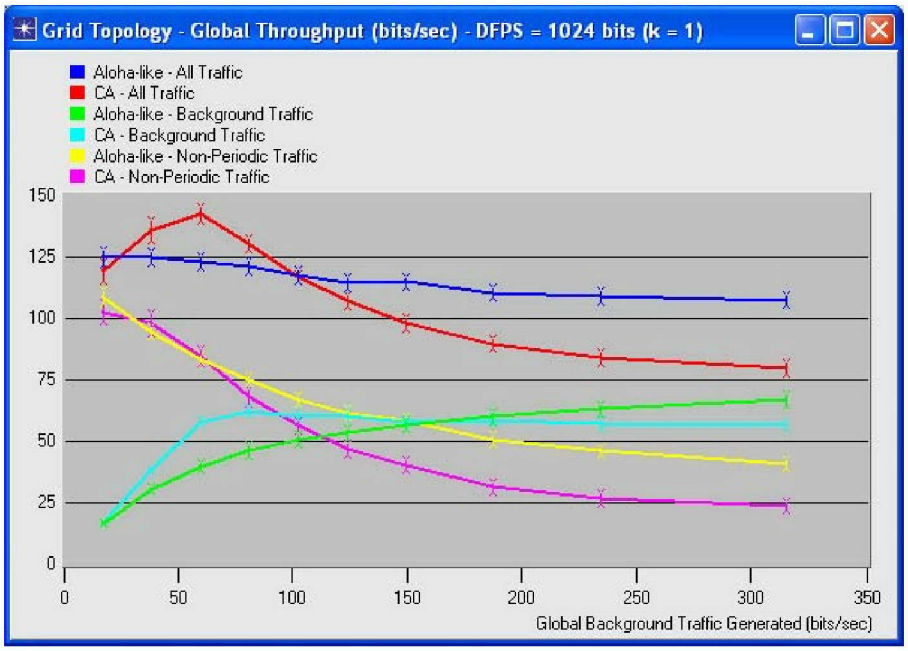

Figure 41. Grid Topology – Load vs Throughput of the different type of traffics. with a DFPS equal to 1024 bits

With the increase of network load, the throughput of the background traffic increases, and the throughput of the non-periodic traffic decreases.

Regarding the end-to-end delay (Figures 42, 43, and 44), the CA/MS mechanism shows a better performance in terms of non-periodic traffic performance. However, it should be reinforced that the Aloha-like MAC scheme does not perform well with a pattern of multiple frames per packet, as is the case with non-periodic traffic.

Although in both cases the performance changes when the network load is increased, the CA/MS seems to have much more sensitivity to the load increase. Despite that, considering the performance of only the non-periodic traffic, the CA/MS scheme has the best performance.

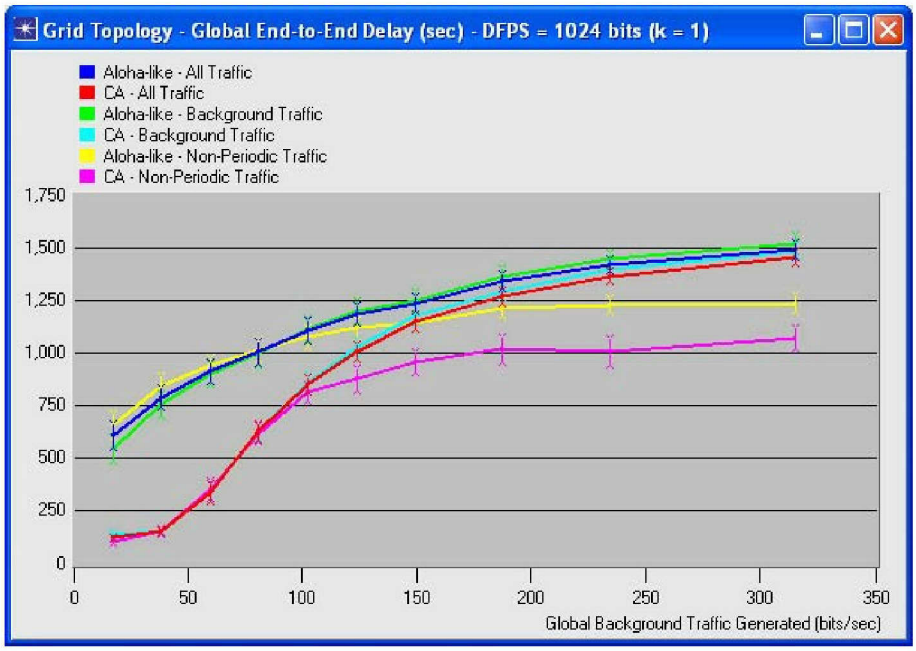

Figure 42. Grid Topology – Load vs End-to-End Delay of the different types of traffic. with a DFPS equal to 1024 bits

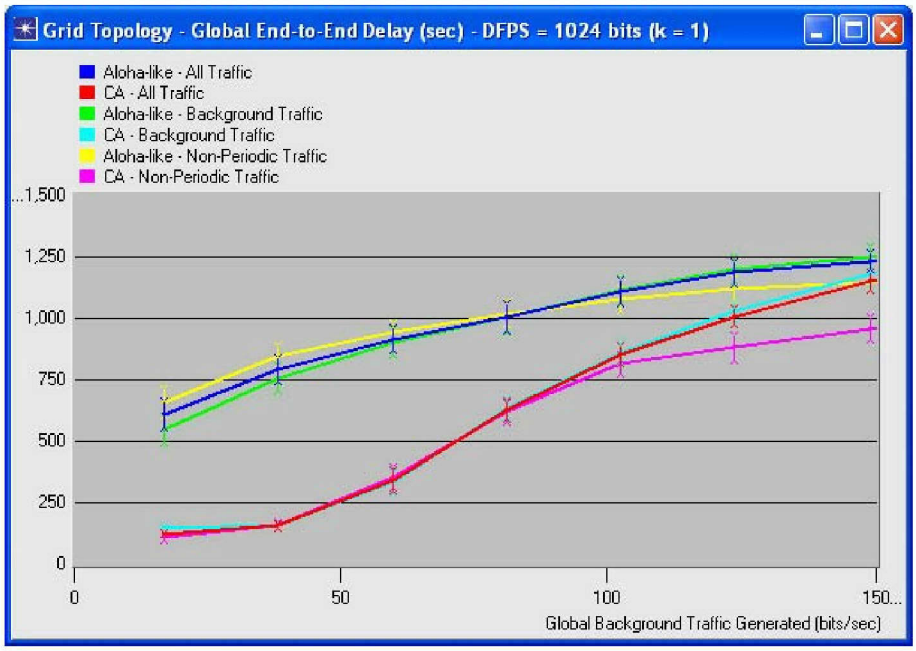

Figure 43. Zoom into the left lower corner of the graph on Figure 42

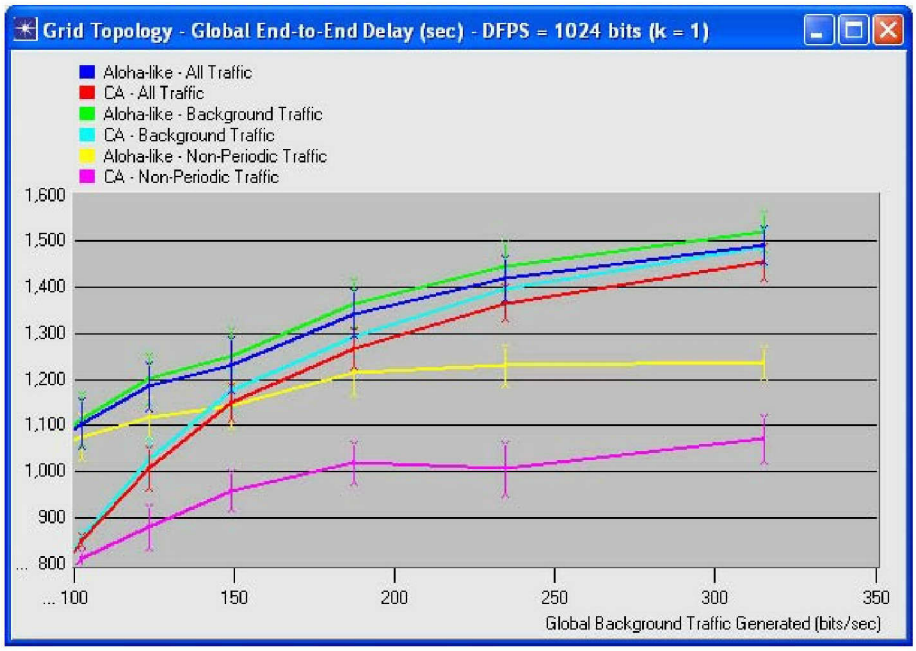

Figure 44. Zoom into the right higher corner of the graph on Figure 42

D. CHAPTER CONCLUSIONS

The results presented and observations made in this chapter are very similar to the results and observations from the tree topology, and can be summarized by the following conclusions:

- The performance of the CA/MS MAC schemes, both throughput and end-to-end delay, is marginally affected by the change in the number of frames per packet.
- The performance of the Aloha-like MAC scheme, both throughput and end-to-end delay, is greatly affected by the change in number of frames per packet.
- Although not fully explored, the Aloha-like MAC scheme appears to have an upper bound for throughput, closely related with the frame size.
- With one frame per packet, the Aloha-like MAC mechanism outperforms clearly the CA/MS MAC scheme in both, throughput and end-to-end delay. This is a result explained by the change in the relative performance of both schemes when the propagation speed of the medium is changed (see Chapter V for more details).
- Regarding the behavior of both schemes when the network is submitted to non-periodic traffic, the CA/MS MAC mechanism shows a better throughput and end-to-end delay performance. However, the nature of the non-periodic traffic introduces multiple frames packets, as already demonstrated, is a situation where the performance of Aloha-like MAC scheme is diminished.

The presentation of the simulation results is completed in this chapter. The following chapter will conclude the thesis, and present recommendations for future work.

VIII. CONCLUSION, RECOMMENDATIONS, AND FUTURE WORKS

A. CONCLUSION AND RECOMMENDATIONS

This thesis studied two alternative MAC mechanisms: 1) a CA/MS scheme, mimicking the wireless aerial radio-based solution, with the exchange of two small control messages in order to reserve the medium, and 2) an uncoordinated Aloha-like MAC scheme, where the node sends the data frame without any prior coordination. Both of the mechanisms considered the key limiting factors that impair the performance of acoustic communications, namely the extremely low propagation speed (1500 m/s), and the limited bandwidth (around 1000 bps)

The introduction (Chapter I)defined the two research questions the thesis endeavored to answer. The study was conducted through modeling and simulation, using the simulation tool, OpNet 10.5.A PL3. The answers are as follows:

- With a configuration of one frame ($k = 1$) per message, the Aloha-like MAC scheme outperforms the CA/MS MAC mechanism. This result is persistent across the topologies considered.
- Considering a fixed frame size, the CA/MS MAC mechanism demonstrates a better end-to-end delay for non-periodic traffic. This result is persistent across the topologies considered.

The detailed conclusions obtained from the simulation results are as follows:

- The CA/MS MAC scheme is not sensitive to variations in number of frames per packet, denoted as "k"; whereas the Aloha-like has great sensitivity to k. With $k = 8$ its performance is worse than the one showed by the CA/MS scheme. However, when k is decreased, the Aloha-like improves its performance, and with $k = 1$, it clearly outperforms the CA/MS scheme, in terms of throughput and end-to-end message delay. This result defies the conventional wisdom that contention based schemes perform better in general, and uncoordinated MAC mechanisms, such as Aloha-like, are only suitable for networks with low loads.
- *Recommendation*: Though the implementations of both protocols in this study can be improved, there is enough evidence to suggest that the trend of the results will not change. That is, the superior performance of Aloha-like schemes seen under the conditions studied will likely maintain under possible protocol optimizations. See the next section for details.

- The results presented in Chapter VI and VII show that the relative performances of the two MAC protocols, in terms of throughput and end-to-end delay, are consistent across the considered topologies.
- *Recommendation:* Although the topologies considered are representative of current UANs implementations, the possibility of denser neighborhoods, especially in the tree topology were not addressed in this study. Such clustering might be expected to degrade the performance of both protocols. It would be interesting to see which protocol scales better with the cluster size.
- The post-simulation results presented in Chapter V shows how the propagation speed of the medium affects the performances of the two protocols. The performance of the Aloha-like MAC scheme does not vary much. The throughput of the CA/MS MAC scheme increases as the propagation speed of the medium increases, and only outperforms the Aloha-like scheme when the propagation speed is much higher than the nominal propagation speed in UANs.
- *Recommendation:* This thesis is insufficient to establish a statistically valid relation between the performance of the CA/MS MAC and the propagation speed of the medium. More work is required in this direction.
- The results suggest that for a fixed frame size, the Aloha-like MAC scheme has a performance upper bound for throughput, and lower bound for end-to-end delay that is dependent upon the frame size.
- *Recommendation:* This study did not look specifically for the relationship between the frame size and the performance of the Aloha-like MAC scheme. Thus, further study is recommended to establish this relationship, in order to determine the optimal frame size for the Aloha-like MAC scheme in the context of UANs.
- Regarding the suitability of Aloha-like MAC schemes when serving non-periodic traffic, it is clear that the characteristics of the non-periodic traffic, with multiple frames per packet, are not the most favorable conditions for the Aloha-like performance.
- *Recommendation:* The Aloha-like implementation should consider variable frame size in order to adapt to different packet size generation, and to take advantage of conditions in which it can deliver its best performance.

B. FUTURE WORK

To the best of the author's knowledge, this is the first study considering an uncoordinated, Aloha-like algorithm as a viable media access control scheme for UANs. The conventional wisdom is that this type of scheme would *only* be appropriate in

networks with light traffic loads. Given the simulation results, this conjecture seems not to hold in UANs. The following are some recommendations for future work in order to explore this subject further:

- Optimize the implementation of both protocols. The performance of the CA/MS scheme might be improved by including implicit ACKs in the RTS sent to the next forwarding node. The performance of the Aloha-like scheme might be improved by removing node's obligation to perform a back-off for every successfully transmitted frame, regardless of whether or not the node has more frames to transmit.
- Improve the fidelity of the OpNet model implementation with the inclusion of a more detailed model of the physical environment of UANs.
- Determine a statistically valid relationship between the performance of the CA/MS scheme and the propagation speed of the medium.
- Study the relationship between the performance of the Aloha-like scheme and the frame size in order to determine the optimal configuration for the protocol in networks with known delay characteristics.
- Carry out performance comparison with other MAC mechanisms, such as the *a priori* channel allocation, which can potentially support full-duplex communication [Gibson 2005b].
- Introduce mobile nodes to determine the ability of the two MAC schemes to perform with mobility introduced, namely to measure the required overhead to handle the link discontinuities and associate a mobile node to the network.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A. PHYSICAL LAYER PROCESS MODEL C-LANGUAGE CODE (OPNET)

```
###############################################################################
                  Process Model Report:  uan_physical_layer
###############################################################################

===============================================================================
                       Process Model Interface Attributes
===============================================================================
-------------------------------------------------------------------------------
                       Interface Attribute: begsim intrpt
-------------------------------------------------------------------------------
Assign Status: hidden
Initial Value  enabled
Data Type:     toggle
Comments:              YES
This attribute specifies whether a 'begin simulation interrupt' is generated for a
processor  module's root process at the start of the simulation.

-------------------------------------------------------------------------------
                         Interface Attribute: doc file
-------------------------------------------------------------------------------
Assign Status: hidden
Initial Value  nd_module
Data Type:     string
Comments:              YES
       This attribute defines the name of the product
       help file which will be displayed when the user invokes help
       for this object.

-------------------------------------------------------------------------------
                       Interface Attribute: endsim intrpt
-------------------------------------------------------------------------------
Assign Status: hidden
Initial Value  disabled
Data Type:     toggle
Comments:              YES
This attribute specifies whether an 'end simulation interrupt' is generated for a
processor  module's root process at the end of the simulation.

-------------------------------------------------------------------------------
                      Interface Attribute: failure intrpts
-------------------------------------------------------------------------------
Assign Status: hidden
Initial Value  disabled
Data Type:     enumerated
Comments:              YES
This attribute specifies whether failure interrupts are generated for a processor
module's root process upon failure of nodes or links in the network model.

-------------------------------------------------------------------------------
                      Interface Attribute: intrpt interval
-------------------------------------------------------------------------------
Assign Status: hidden
Initial Value  disabled
Data Type:     toggle double
Comments:              YES
This attribute specifies how often regular interrupts are scheduled for the root process
of a processor module.

-------------------------------------------------------------------------------
                         Interface Attribute: priority
-------------------------------------------------------------------------------
Assign Status: hidden
Initial Value  0
```

```
Data Type:     integer
Comments:              YES
       This attribute is used to determine the execution order of events that are
scheduled to occur at the same simulation time.

--------------------------------------------------------------------------------
                    Interface Attribute: recovery intrpts
--------------------------------------------------------------------------------
Assign Status: hidden
Initial Value  disabled
Data Type:     enumerated
Comments:              YES
This attribute specifies whether recovery interrupts are scheduled for the processor
module's root process upon recovery of nodes or links in the network model.

--------------------------------------------------------------------------------
                         Interface Attribute: subqueue
--------------------------------------------------------------------------------
Assign Status: hidden
Initial Value  (...)
Data Type:     compound
Comments:              YES
This operation attribute permits the addition and deletion of subqueues within the queue
module.

--------------------------------------------------------------------------------
                       Interface Attribute: super priority
--------------------------------------------------------------------------------
Assign Status: hidden
Initial Value  disabled
Data Type:     toggle
Comments:              YES
       This attribute is used to determine the execution order of events that are
scheduled to occur at the same simulation time.



================================================================================
                                  Header Block
================================================================================
#include "uan_support.h"
#include <math.h>

/* Define a small value (= 1 psec), which will be used to recover from double
   arithmetic precision losts while doing time related precision sensitive
   computations. */
#define        PRECISION_RECOVERY                    0.000000000001

/* Define the module streams. They need to match the stream numbers in the node
   editor */
#define OUTPUT_STREAM_TO_MAC_LAYER 0
#define INPUT_STREAM_FROM_MAC_LAYER 0
#define INPUT_STREAM_EXTERNAL_NODES 1

/* Transition Macros */
#define FRAME_RECEIVED (op_intrpt_type() == OPC_INTRPT_STRM && op_intrpt_strm() ==
INPUT_STREAM_EXTERNAL_NODES)
#define FRAME_FROM_MAC_LAYER (op_intrpt_type() == OPC_INTRPT_STRM && op_intrpt_strm() ==
INPUT_STREAM_FROM_MAC_LAYER)
#define RECEIVER_ON (op_intrpt_type() == OPC_INTRPT_REMOTE && op_intrpt_code() ==
UwnE_Receiver_On)
#define RECEIVER_OFF (op_intrpt_type() == OPC_INTRPT_REMOTE && op_intrpt_code() ==
UwnE_Receiver_Off)

// Utility macros
#define PHYSICAL_LAYER "Physical Layer"
#define MAC_LAYER "MAC Layer"
#define TXTIME(size) (((double)size) / dataRate)

// Structure for characterize the reachable nodes
typedef struct UanT_Reachable_Neighbors
```

```
{
        char name[25];                          // Name of the reachable node
        double propagationDelay; // Propagation delay for this reachable node

} UanT_Reachable_Neighbors;

static void defineReachableNodes(void);
static double getDistance (double myXpos, double myYpos,
                                double otherXpos, double otherYpos);
static void initialization();
================================================================================
                              State Variable Block
================================================================================
/* The maximum distance the signal's node can reach */
double  \range;

/* Handle for this object (module) */
Objid   \myObjectID;

/* Handle for the parent object of this module (node) */
Objid   \myNodeObjectID;

/* Handle for the grand parent object this module (network) */
Objid   \myNetworkObjectID;

/* The physical layer error, as a percentage of received frames */
double  \errorRate;

/* The propagation speed of the medium */
double  \propagationSpeed;

/* The list of the reachable nodes (within the range of this node) */
List*   \reachableNeighborsList;

/* Handle to the MAC Layer object of this module */
Objid   \myMacLayer;

/* The name of the node object where this module is integrated */
char    \myName[25];

/* The data rate defined for the channel */
double  \dataRate;

================================================================================
                            Temporary Variable Block
================================================================================
/* The current packet being processed */
Packet* currentFrame = OPC_NIL;

// Copy to send to the reachable nodes
Packet* copyFrame;

// Temporary variables for handling the act of sending to the reachable nodes
int idx;
UanT_Reachable_Neighbors* reachableNeighbor;
Objid remoteNode;
Objid remotePhysicalLayer;
================================================================================
                                 Function Block
================================================================================
/* ###########################################################################
Utility function for populate the list of reachable nodes
*/
static void defineReachableNodes()
{
        int idx;
        int nrNodes;
        Objid externNode;
        double myXposition;
        double myYposition;
        double externXposition;
```

```
	double externYposition;
	double distance;
	UanT_Reachable_Neighbors* reachableNeighbor;

	FIN (defineReachableNodes());

	// Our geographical position
	op_ima_obj_attr_get (myNodeObjectID, "x position", &myXposition);
	op_ima_obj_attr_get (myNodeObjectID, "y position", &myYposition);

	// Number of nodes in the network
	nrNodes = op_topo_child_count (myNetworkObjectID, OPC_OBJMTYPE_NODE);

	/* Loop through all the nodes in the network and check if they are within
	   our range */
	for (idx = 0; idx < nrNodes; idx++)
	{
		// Get the idx node in the network
		externNode = op_topo_child (myNetworkObjectID, OPC_OBJMTYPE_NODE, idx);

		// Do not perform the algorithm for my own node
		if (externNode != myNodeObjectID)
		{
			// Get the geographical position of the idx neighbor node
			op_ima_obj_attr_get (externNode, "x position", &externXposition);
			op_ima_obj_attr_get (externNode, "y position", &externYposition);

			// Calculate the distance
			distance = getDistance (myXposition, myYposition,
						externXposition, externYposition);

			// Add the node to the list only if it is within our range
			if (distance <= range)
			{
				// Allocate memory for the new list element
				reachableNeighbor = (UanT_Reachable_Neighbors*)
					op_prg_mem_alloc (sizeof(UanT_Reachable_Neighbors));

				// Add node's name
				op_ima_obj_attr_get (externNode, "name",
					reachableNeighbor->name);

				// Add node's propagation delay
				reachableNeighbor->propagationDelay =
					distance / propagationSpeed;

				// Add this element to the list
				op_prg_list_insert(reachableNeighborsList,
					reachableNeighbor, OPC_LISTPOS_TAIL);
			}

		}

	} // end for

	FOUT;
}


/* ##############################################################################
Accept the geographical position of two nodes in meters and calculates the
distance between them
*/
static double getDistance (double myXpos, double myYpos, double otherXpos,
				double otherYpos)
{
	FIN (getDistance (myXpos, myYpos, otherXpos, otherYpos));

	FRET (sqrt (pow((myXpos - otherXpos), 2.0) + pow((myYpos - otherYpos), 2.0)));
}
```

```
/* ###########################################################################
 Initializes the Physical Layer process model
*/
static void initialization()
{
	Objid macLayer;

	FIN (initialization());

	macLayer = op_id_from_name (myNodeObjectID, OPC_OBJTYPE_QUEUE, MAC_LAYER);

	// Retrieve the attributes needed and populate the state variables
	op_ima_obj_attr_get (macLayer, "Range", &range);
	op_ima_obj_attr_get (macLayer, "Error Rate", &errorRate);
	op_ima_sim_attr_get (OPC_IMA_DOUBLE, "Propagation Speed", &propagationSpeed);
	op_ima_obj_attr_get (macLayer, "Outbound Channel Data Rate", &dataRate);
	op_ima_obj_attr_get (myNodeObjectID, "name", myName);

	// Create the list for storing the reachable nodes
	reachableNeighborsList = op_prg_list_create();

	FOUT;
}
===========================================================================
                    Enter Execs for the forced state "send"
===========================================================================
/* This state process the reception of the frame from our MAC layer, and
   delivers it to all reachable nodes */

// Retrieving the frame from the stream used by our link layer to sent us frames
currentFrame = op_pk_get(INPUT_STREAM_FROM_MAC_LAYER);

/* We do not need to signal the beginning of transmission, because the MAC layer
assumes the beginning of transmission as the moment it sends the frame to us.
However we need to signalling the end of transmission with the correct
transmission delay */
op_intrpt_schedule_remote (op_sim_time() +
	TXTIME (op_pk_total_size_get (currentFrame)), UwnE_Transmitter_Off,
	myMacLayer);

/* Loop through the reachable neighbors list and send the begin and end of
reception, and the frame itself */
for (idx = 0; idx < op_prg_list_size (reachableNeighborsList); idx++)
{
	// Handle to the idx element of the list
	reachableNeighbor = (UanT_Reachable_Neighbors*)
		op_prg_list_access (reachableNeighborsList, idx);

	// Handle to the idx neighbor
	remoteNode = op_id_from_name(op_topo_parent(op_topo_parent(op_id_self())),
		OPC_OBJTYPE_NODE_FIX, reachableNeighbor->name);

	// Print to debug mode
	printf("\n#%s# The physical layer is sending begin of reception, the frame,"
		   "and end of reception events to %s", myName,
		   reachableNeighbor->name);

	// Handle to the physical layer module of the idx neighbor
	remotePhysicalLayer = op_id_from_name(remoteNode, OPC_OBJTYPE_PROC,
		PHYSICAL_LAYER);

	/* Scheduling the beginning of reception for the idx neighbor with the
	   correct propagation delay */
	op_intrpt_schedule_remote (op_sim_time() +
		reachableNeighbor->propagationDelay, UwnE_Receiver_On,
		remotePhysicalLayer);

	/* Scheduling the deliver of the frame to the idx neighbor, but before copy
	   the frame. A sent frame changes owner, and without that we could only
	   sent one */
```

```
        copyFrame = op_pk_copy(currentFrame);
    op_pk_deliver_delayed(copyFrame, remotePhysicalLayer,
                INPUT_STREAM_EXTERNAL_NODES,
                TXTIME (op_pk_total_size_get (currentFrame)) +
                reachableNeighbor->propagationDelay);

        /* Schedule the end of reception for the same time that the frame was sent.
           The time added is 1 psec to guarantee the end of reception after the
           actual frame reception */
        op_intrpt_schedule_remote (op_sim_time() +
                TXTIME (op_pk_total_size_get (currentFrame)) +
                reachableNeighbor->propagationDelay + PRECISION_RECOVERY,
                UwnE_Receiver_Off, remotePhysicalLayer);

}

// After sent a copy to all nodes, destroy the frame received from the MAC layer
op_pk_destroy(currentFrame);
--------------------------------------------------------------------------------
================================================================================
                      Exit Execs for the forced state "send"
================================================================================
NONE
--------------------------------------------------------------------------------
```

APPENDIX B. MAC PROCESS MODEL C-LANGUAGE CODE (OPNET)

```
################################################################################
                        Process Model Report:  uan_mac_layer
################################################################################

================================================================================
                              Process Model Attributes
================================================================================
--------------------------------------------------------------------------------
                               Attribute: MAC Address
--------------------------------------------------------------------------------
Data Type:     integer
Comments:
        The MAC address of the current node.

--------------------------------------------------------------------------------
                               Attribute: Frames Size
--------------------------------------------------------------------------------
Data Type:     compound
Comments:
        Specifies the size of the frames used in the model

--------------------------------------------------------------------------------
                            Attribute: Inbound Channels
--------------------------------------------------------------------------------
Data Type:     compound
Comments:
        Number, identification and characteristics of the channels

--------------------------------------------------------------------------------
                                  Attribute: Range
--------------------------------------------------------------------------------
Data Type:     double
Comments:
        Specifies the value that the model uses to define the range that the node's signal
can reach. It is a discrete type model, that is, the node can reach all nodes within the
range defined, and can not reach a node that is far away more than exactly the value
defined.

--------------------------------------------------------------------------------
                           Attribute: Frame Transmission
--------------------------------------------------------------------------------
Data Type:     compound

--------------------------------------------------------------------------------
                               Attribute: Error Rate
--------------------------------------------------------------------------------
Data Type:     double
Comments:
        The error rate of all channels.  The model determines if the frame has error with
the result of a uniform distribution between 0 and 1. If the result is less that the
defined error rate attribute, then the frame is discarded.

--------------------------------------------------------------------------------
                            Attribute: Destination Node
--------------------------------------------------------------------------------
Data Type:     integer
Comments:
        The MAC address of the single node to where all the traffic of this node will be
forwarded

--------------------------------------------------------------------------------
```

```
                    Attribute: Outbound Channel Data Rate
--------------------------------------------------------------------------------
Data Type:     double
Comments:
        The data rate of the single sending channel

--------------------------------------------------------------------------------
                        Attribute: Outbound Channel ID
--------------------------------------------------------------------------------
Data Type:     integer

--------------------------------------------------------------------------------
                         Attribute: UAN Type Of Node
--------------------------------------------------------------------------------
Data Type:     integer

--------------------------------------------------------------------------------
                            Attribute: Contention
--------------------------------------------------------------------------------
Data Type:     compound



================================================================================
                       Process Model Interface Attributes
================================================================================
--------------------------------------------------------------------------------
                        Interface Attribute: begsim intrpt
--------------------------------------------------------------------------------
Assign Status: hidden
Initial Value  enabled
Data Type:     toggle
Comments:              YES
        This attribute specifies whether a 'begin simulation interrupt' is generated for a
processor  module's root process at the start of the simulation.

--------------------------------------------------------------------------------
                          Interface Attribute: doc file
--------------------------------------------------------------------------------
Assign Status: hidden
Initial Value  nd_module
Data Type:     string
Comments:              YES
        This attribute defines the name of the product
        help file which will be displayed when the user invokes help
        for this object.

--------------------------------------------------------------------------------
                        Interface Attribute: endsim intrpt
--------------------------------------------------------------------------------
Assign Status: set
Initial Value  enabled
Data Type:     toggle
Comments:              YES
        This attribute specifies whether an 'end simulation interrupt' is generated for a
processor  module's root process at the end of the simulation.

--------------------------------------------------------------------------------
                      Interface Attribute: failure intrpts
--------------------------------------------------------------------------------
Assign Status: hidden
Initial Value  disabled
Data Type:     enumerated
Comments:              YES
        This attribute specifies whether failure interrupts are generated for a processor
module's root process upon failure of nodes or links in the network model.

--------------------------------------------------------------------------------
                      Interface Attribute: intrpt interval
--------------------------------------------------------------------------------
Assign Status: hidden
```

Initial Value disabled
Data Type: toggle double
Comments: YES
        This attribute specifies how often regular interrupts are scheduled for the root process of a processor module.

Interface Attribute: priority

Assign Status: hidden
Initial Value 0
Data Type: integer
Comments: YES
        This attribute is used to determine the execution order of events that are scheduled to occur at the same simulation time.

Interface Attribute: recovery intrpts

Assign Status: hidden
Initial Value disabled
Data Type: enumerated
Comments: YES
        This attribute specifies whether recovery interrupts are scheduled for the processor module's root process upon recovery of nodes or links in the network model.

Interface Attribute: subqueue

Assign Status: hidden
Initial Value (...)
Data Type: compound
Comments: YES
        This operation attribute permits the addition and deletion of subqueues within the queue module.

Interface Attribute: super priority

Assign Status: hidden
Initial Value disabled
Data Type: toggle
Comments: YES
        This attribute is used to determine the execution order of events that are scheduled to occur at the same simulation time.

==========

Process Model Global Attributes

==========

Attribute: UAN_Network_Mode

Data Type: integer

Attribute: Data_Frame_Payload_Size

Data Type: integer

Attribute: Propagation_Speed

Data Type: double
Comments:
        The propagation speed of the medium

==========

```
                                    Header Block
================================================================================
#include <math.h>

#include <string.h>
#include <prg_list_funcs.h>
#include "uan_support.h"

#define LOCAL_PHYSICAL_LAYER "Physical Layer"
#define MAC_LAYER "MAC Layer"

/* Statistical and Internal Streams. They should match the stream numbers in the node
editor */
#define INPUT_STREAM_FROM_UPPER_LAYER_BACKGROUND 0
#define INPUT_STREAM_FROM_UPPER_LAYER_NON_PERIODIC 1
#define INPUT_STREAM_FROM_PHYSICAL_LAYER 2
#define OUTPUT_STREAM_TO_UPPER_LAYER 0
#define OUTPUT_STREAM_TO_PHYSICAL_LAYER 1
#define MAX_NUMBER_NODES 20

// Define user defined interrupts
#define FRAME_ARRIVAL 10

/* Special value indicating that the number of back-off slots are   */
/* not determined yet.
               */
#define        BACKOFF_SLOTS_UNSET                  -1.0

/* Define a small value (= 1 psec), which will be used to recover   */
/* from double arithmetic precision losts while doing time related  */
/* precision sensitive computations.
       */
#define        PRECISION_RECOVERY                   0.000000000001

/* Transition Macros */
#define FRAME_FROM_PHYSICAL_LAYER (op_intrpt_type() == OPC_INTRPT_STRM &&
op_intrpt_strm() == INPUT_STREAM_FROM_PHYSICAL_LAYER)
#define PKT_FROM_UPPER_LAYER (op_intrpt_type() == OPC_INTRPT_STRM && op_intrpt_strm() ==
INPUT_STREAM_FROM_UPPER_LAYER)

/* After receiving a stream interrupt, we need to switch states from       */
/* idle to defer or transmit if there is a frame to transmit and the       */
/* receiver is not busy
                              */
/* If a frame is received indicating that the STA should scan, all bets */
/* are off, and the STA moves into the scan state to look for other APs */
#define NEED_TO_TRANSMIT      (flags->packet_to_send == OPC_TRUE  || flags-
>fragments_to_send == OPC_TRUE ||\
                                                    frameTypeToSend != UanE_None)
/*

                              */

/*
When we have a frame to transmit, we move to transmit state if the medium was idle for at
least a DIFS time,
otherwise we go to defer state.
*/
#define MEDIUM_IS_IDLE  ((op_sim_time() - navTime + PRECISION_RECOVERY >= difsDuration)
&& \
                           flags->receiver_busy == OPC_FALSE && \
                                                (op_sim_time() - receiverIdleTime +
PRECISION_RECOVERY >= difsDuration))

/* Backoff is performed based on the value of the backoff flag.                  */
#define PERFORM_BACKOFF                       (flags->perform_backoff == OPC_TRUE)

/* Macros that check the change in the busy status of the receiver.        */
#define RECEIVER_BUSY_HIGH (interruptType == OPC_INTRPT_STAT && interruptCode <
TRANSMITTER_BUSY_INSTAT && !flags->collision)
```

```
#define RECEIVER_BUSY_LOW      (((interruptType == OPC_INTRPT_STAT && interruptCode <
TRANSMITTER_BUSY_INSTAT) ||                  \
                                                              (interruptType ==
OPC_INTRPT_STRM && interruptCode != INPUT_STREAM_FROM_UPPER_LAYER)) &&     \
                                    !flags->receiver_busy)

#define PERFORM_TRANSMIT               (BACKOFF_COMPLETED && MEDIUM_IS_IDLE &&
frameTypeToSend == UanE_Rts)

#define BACK_TO_DEFER  (FRAME_RCVD || (BACKOFF_COMPLETED && NEED_TO_TRANSMIT))

/* If the frame is received appropriate response will be transmitted    */
/* provided the medium is considered to be idle
*/
#define FRAME_RCVD                          (interruptType == OPC_INTRPT_REMOTE &&
bad_packet_rcvd == OPC_FALSE && \
                                                              interruptCode ==
UwnE_Receiver_On)

/* Need to start transmitting frame once the backoff (self intrpt)         */
/* completed
                              */
#define BACKOFF_COMPLETED              (interruptType == OPC_INTRPT_SELF && interruptCode
== UwnE_Backoff_Elapsed && \
                                                              (flags->receiver_busy ==
OPC_FALSE || flags->forced_bk_end == OPC_TRUE))

/* After deferring for either collision avoidance or interframe gap        */
/* the channel will be available for transmission.
       */
#define DEFERENCE_OFF  (interruptType == OPC_INTRPT_SELF && \
                                                              interruptCode ==
UwnE_Deference_Off && \
                                                              flags->receiver_busy ==
OPC_FALSE)

/* Skip backoff if no backoff is needed
           */
#define TRANSMIT_FRAME                 (!PERFORM_BACKOFF)

/* Change state to Defer from Frm_End, if the input buffers are not empty or a frame
needs   */
/* to be retransmitted or the station has to respond to some frame.
                        */
#define FRAME_TO_TRANSMIT (flags->packet_to_send == OPC_TRUE || frameTypeToSend !=
UanE_None || \
                                                        flags->fragments_to_send == OPC_TRUE
|| flags->backoff_required == OPC_TRUE)

/* When the contention window period is over then we go to IDLE state      */
/* if we don't have another frame to send at that moment. If we have       */
/* one then we go to TRANSMIT state if we did not sense any activity       */
/* on our receiver for a period that is greater than or equal to DIFS      */
/* period; otherwise we go to DEFER state to defer and back-off before     */
/* transmitting the new frame.
                        */
#define BACK_TO_IDLE                   (BACKOFF_COMPLETED && !NEED_TO_TRANSMIT)

//#define SEND_NEW_FRAME_AFTER_CW     (CW_COMPLETED && flags->packet_to_send == OPC_TRUE
&& MEDIUM_IS_IDLE)

#define DEFER_AFTER_CW                 (CW_COMPLETED && flags->packet_to_send == OPC_TRUE
&& !MEDIUM_IS_IDLE)

/* Need to retransmit frame if there is a frame timeout and the         */
/* required frame is not (successfully) received
       */
#define FRAME_TIMEOUT           ((interruptType == OPC_INTRPT_SELF && interruptCode ==
UwnE_Frame_Timeout) /*|| \
                                                              bad_cts_to_self_rcvd ==
OPC_TRUE*/)
```

```
#define RESUME_TIMEOUT    (op_intrpt_type() == OPC_INTRPT_SELF && op_intrpt_code() ==
UanE_Resume_Timeout)

/* Issue a transmission complete stat once the packet has successfully     */
/* been transmitted from the source station
       */
#define TRANSMISSION_COMPLETE (interruptType == OPC_INTRPT_REMOTE && \
                                                              op_intrpt_code () ==
UwnE_Transmitter_Off)

/* After transmission the station will wait for a frame response for    */
/* Data and Rts frames.
                          */
#define FRM_END_TO_IDLE                       (!FRAME_TO_TRANSMIT && !EXPECTING_FRAME)

#define FRM_END_TO_DEFER              (!EXPECTING_FRAME && FRAME_TO_TRANSMIT)

/* Expecting frame response   after data or Rts transmission                          */
#define EXPECTING_FRAME                       (expectedFrameType != UanE_None &&
expectedFrameType != UanE_None_Transit)

#define EXPECTING_DATA (op_intrpt_type() == OPC_INTRPT_SELF && op_intrpt_code() ==
FRAME_ARRIVAL && \
                        expectedFrameType == UanE_Data)

#define EXPECTING_ACK (op_intrpt_type() == OPC_INTRPT_SELF && op_intrpt_code() ==
FRAME_ARRIVAL && \
                        expectedFrameType == UanE_Ack)

#define EXPECTING_CTS (op_intrpt_type() == OPC_INTRPT_SELF && op_intrpt_code() ==
FRAME_ARRIVAL && \
                        expectedFrameType == UanE_Cts)

#define TXTIME(size)   (((double) size) / outboundChannelDataRate)

/* Global variables for collect the end simulation stats. The index corresponds to
the address of each node. For example, the values of sensor1 is stored on array
element with index 1. We consider a maximum of 20 possible nodes */

// Global variables Collect the sum of the bits generated by the nodes
double bitsGeneratedByNode[MAX_NUMBER_NODES];
int packetsGeneratedByNode[MAX_NUMBER_NODES];
int packetsInQueueByNode[MAX_NUMBER_NODES];
int packetsDroppedByNode[MAX_NUMBER_NODES];
double backoffSlotsByNode[MAX_NUMBER_NODES];
int retransmissionAttemptsByNode[MAX_NUMBER_NODES];
int framesCollidedByNode[MAX_NUMBER_NODES];

char nodesName[MAX_NUMBER_NODES][10];

double lastTransmittedPacket[20];
double lastReceivedPacket[20];
/* Data structures */

typedef enum UanT_Network_Mode
{
       UanE_Aloha_Alike,
       UanE_Contention_Based,
} UanT_Network_Mode;


typedef struct UanT_Frame_Fields
{
       UanT_Mac_Frame_Type frameType;
       int numberOfFragments;
       int retry;
       OpT_Packet_Id dataPacketID;
       int fragmentNumber;
       double senderTime;
       double reservedDuration;    /* NAV */
```

```
        int originatorAddress;       /* original sender */
        int senderAddress;       /* final destination */
        int receiverAddress;       /* sender */
        double dataRate;
        Boolean ackFragments [16];
        int typeOfTraffic;
} UanT_Frame_Fields;



/* Define a structure to maintain data fragments received by each     */
/* station for the purpose of reassembly (or defragmentation)                */
typedef struct UwnT_Mac_Defragmentation_Buffer_Entry
        {
        int                    tx_station_address    ;/* Store the station address of
transmitting station          */
        OpT_Packet_Id  dataPacketID;
        int  receivedNumberOfFragments;
        Boolean sentHigherLayer;
        Boolean ackFragments[16];
        double         time_rcvd                    ;/* Store time the last fragment for
this frame was received       */
        Sbhandle       reassembly_buffer_ptr ;/* Store data fragments for a particular
packet                        */
        } UwnT_Mac_Defragmentation_Buffer_Entry;

/* Define a structure to maintain a copy of each unique data frame      */
/* received by the station. This is done so that the station can           */
/* discard any additional copies of the frame received by it.                    */
typedef struct UwnT_Mac_Duplicate_Buffer_Entry
        {
        int         tx_station_address;  /* store the station address of transmitting
station */
        int            sequence_id              ;  /* rcvd packet sequence id
                                    */
        int                fragment_number      ;  /* rcvd packet fragment number
        */
        } UwnT_Mac_Duplicate_Buffer_Entry;


/* Define a structure to hold the information about the fragments of a data packet to
send */
typedef struct UanT_Mac_Fragmentation_List_Element
{
        int fragmentNumber;
        int numberOfFragments;
        int typeOfTraffic;
        Boolean transmitted;
        OpT_Packet_Size fragmentSize;
        Packet* fragment;
}UanT_Mac_Fragmentation_List_Element;

/* Defined type to support the channel caracterization */
typedef struct UwnT_Channel
{
        int channelID;            /* Unique identification of the channel */
        int dataRate;             /* The channel's data rate */
        double receiverIdleTime;      /* */
        double receiverEndTime;
} UwnT_Channel;

/* This structure contains all the flags used in this process model to determine   */
/* various conditions as mentioned in the comments for each flag
        */
typedef struct UwnT_Flags
        {
        Boolean        packet_to_send; /* Flag to check when station needs to transmit.
        */
        Boolean        fragments_to_send;
        Boolean     backoff_required;            /* Backoff flag is set when either the
collision is          */
```

```
        Boolean         perform_backoff;
                                        /* inferred or the channel switched from busy to
idle    */
        Boolean         rts_sent;               /* Flag to indicate that whether the RTS for
this            */
        Boolean         rcvd_bad_packet;        /* Flag to indicate that the received packet
is bad          */
    Boolean         receiver_busy;              /* Set this flag if receiver busy stat is
enabled         */
    Boolean         transmitter_busy;  /* Set this flag if we are transmitting something.
        */
        Boolean         immediate_xmt;          /* Set this flag if the new frame can be
transmitted     */
                                                                /* without deferring.
                                                        */
        Boolean         forced_bk_end;          /* Special case: resume with completion of
back-off (or    */
                                                                /* CW) period
regardless of receiver's status.                        */
        Boolean         nav_updated;            /* Indicates a new NAV value since the last
time when       */
                                                                /* self interrupt is
scheduled for the end of deference.*/
        Boolean         collision;                      /* Set this flag if a channel became
busy while another      */
                                                                /* one busy.
                                                                */
        } UwnT_Flags;

typedef enum UanT_Type_Of_Node
{
        UanE_Sensor_Node = 0,
        UanE_Relay_Node = 1,
        UanE_Gateway_Node = 2
} UanT_Type_Of_Node;


/* Define interrupt codes for generating handling interrupts
        */
/* indicating changes in deference, frame timeout which infers
        */
/* that the collision has occurred, random backoff and transmission
        */
/* completion by the physical layer (self interrupts).
                */
typedef enum UwnT_Mac_Intrpt_Code
        {
        UwnE_Deference_Off,     /* Deference before frame transmission
        */
        UwnE_Frame_Timeout,     /* No frame rcvd in set duration (infer collision)   */
        UanE_Resume_Timeout,
        UwnE_Backoff_Elapsed,   /* Backoff done before frame transmission
        */
        UwnE_CW_Elapsed,                /* Backoff done after successful frame transmission
        */
                                                        /* current AP.
                                                */
        } UwnT_Mac_Intrpt_Code;


/* This structure contains the destination address to which the received */
/* data packet needs to be sent and the contents of the received packet  */
/* from the higher layer.
                        */
typedef struct UanT_High_Layer_List_Elem
        {
        double  time_rcvd;              /* Time packet is received by the higher layer
        */
        int typeOfTraffic;
        Packet* pkptr;                          /* store packet contents
                        */
```

```
        } UanT_High_Layer_List_Elem;

typedef enum UanT_Type_Of_Traffic
{
        UanE_Background_Traffic,
        UanE_Non_Periodic_Traffic
} UanT_Type_Of_Traffic;


/* Function prototypes */
static void high_layer_packet_drop (Packet* highLayerCurrPkt);
static void high_layer_packet_enqueue (Packet* highLayerCurrPktr, int traffic);
static void mac_error (const char* msg1, const char* msg2, const char* msg3);
static void physical_layer_data_arrival (void);
static void schedule_deference ();
static void higher_layer_data_arrival (void);
static void interrupts_process (void);
static void mac_rcv_channel_status_update (int channel_id);
static void frame_discard (void);
static void data_process (Packet * seg_pkptr, UanT_Frame_Fields* rcvdDummyHeader);
static void accepted_frame_stats_update (Packet* seg_pkptr, int address, int traffic);
static void frame_transmit ();
static double calculate_NAV_toSend(void);
static void findDestinationNodeName(void);
static OpT_Boolean errorFrame(void);
static void prepare_Ack_to_send (void);
static void prepare_Cts_to_send (void);
static void prepare_Rts_to_send (void);
static void prepare_data_frame_to_send (void);
static void modelInitialization(void);
static void build_packet_fragments_list(void);
static void updateControlTrafficStats(double, double );
static void updateDataTrafficStats(double, double );
static void printStateVariables(void);
static char* frameType(int type);
static void record_final_stats(void);
================================================================================
                              State Variable Block
================================================================================
/* The object ID of the surrounding module */
Objid   \myObjectID;

/* The object ID of the surrounding node */
Objid   \myNodeObjectID;

/* Logging information if the packet is dropped due to higher layer  */
/* queue overflow.                                                   */
Log_Handle     \configLogHandle;

/* Simulation log handle for configuration related messages. */
Log_Handle     \dropPacketLogHandle;

/* This is to make sure that the entry is written only once.        */
Boolean \dropPacketEntryLogFlag;

/* Range of node's signal. The default value is 1500 m. */
double  \range;

/* The node's address. */
int     \myAddress;

/* Total number of channels that the node uses */
int     \numberOfChannels;

/* Holds all the information that characterizes each channel */
UwnT_Channel   \inboundChannels;

/* List or queue that holds all the information about the channels available on this node
*/
List*   \channels_ptr;
```

```
/* The size of an acknowledgement frame */
int     \sizeACK;

/* The size of a Clear-to-Send frame */
int     \sizeCTS;

/* The size of a Request-to-Send frame */
int     \sizeRTS;

/* The size of a data frame header */
int     \sizeDataFrameHeader;

/* The size of a MAC Service Data Unit. The maximum payload size. */
int     \sizeDataFramePayload;

/* Specifies the maximum number of attempts to retransmmit a frame before give up */
int     \maxRetransmissionAttempts;

/* The flags used in this process model */
UwnT_Flags*    \flags;

/* This buffer contains the fragments received from            */
/* remote station and maintains following information         */
/* for each fragment:
*/
/* 1. remote station address
*/
/* 2. time the last fragment was received                     */
/* 3. reassembly buffer                                                */
List*   \defragmentationListPtr;

/* Common reassembly buffer used for segment containing the entire   */
/* original packet.                                                    */
Sbhandle       \commonRSMbufPtr;

/* Recording number of packets received from the      */
/* higher layer                                                      */
Stathandle     \packetLoadHandle;

/* Reporting the packet size arrived from higher layer.      */
Stathandle     \bitsLoadHandle;

/* Number of backoff slots before transmission. */
Stathandle     \backoffSlotsHandle;

/* Data Traffic sent by the station    */
Stathandle     \dataTrafficSentHandle;

/* Data Traffic received by the station */
Stathandle     \dataTrafficRcvdHandle;

/* Data Traffic sent  by the station in bits  */
Stathandle     \dataTrafficSentHandleInBits;

/* Data Traffic received by the station in bits      */
Stathandle     \dataTrafficRcvdHandleInBits;

/* Control Traffic (Rts,Cts or Ack) sent by the station      */
Stathandle     \ctrlTrafficSentHandle;

/* Control Traffic (Rts,Cts or Ack) received by the station in packets      */
Stathandle     \ctrlTrafficRcvdHandle;

/* Control Traffic (Rts,Cts or Ack) sent by the station in bits      */
Stathandle     \ctrlTrafficSentHandleInBits;

/* Control Traffic (Rts,Cts or Ack) received by the station in bits  */
Stathandle     \ctrlTrafficRcvdHandleInBits;

/* Keep track of the dropped packet by the higher layer queue due    */
/* to the overflow of the buffer.                                      */
```

```
Stathandle     \dropPacketHandle;

/* Keep track of the dropped packet by the higher layer queue due    */
/* to the overflow of the buffer.                                    */
Stathandle     \dropPacketHandleInBits;

/* Keep track of the number of retransmissions before the packet was        */
/* successfully transmitted.                                          */
Stathandle     \retransHandle;

/* Keep tracks of the delay from the time the packet is received                */
/* from the higher layer to the time it is transmitted by the station.      */
Stathandle     \mediaAccessDelay;

/* Handle for the end to end delay statistic that is recorded for     */
/* the packets that are accepted and forwarded to the higher layer.  */
Stathandle     \eteDelayHandle[MAX_NUMBER_NODES];

/* Statistic for network allocation vector    */
Stathandle     \channelReservHandle;

/* Keep track of the number of data bits sent to the higher layer.   */
Stathandle     \throughputHandle;

/* Handle for global end-to-end delay statistic. */
Stathandle     \globalETEdelayHandle;

/* Handle for global WLAN load statistic. */
Stathandle     \globalLoadHandle;

/* Handle for global WLAN throughput statistic.              */
Stathandle     \globalThroughputHandle;

/* Handle for global dropped higher layer data statistic.    */
Stathandle     \globalDroppedDataHandle;

/* Handle for global media access delay statistic.    */
Stathandle     \globalMACdelayHandle;

/* Random number of backoff slots determined by a uniform distribution */
double  \backoffSlots;

/* Incremented each time a frame was unsuccessful in transmission */
int     \retryCount;

/* Make copy of the transmit frame before transmission */
Packet *       \transmitFrameCopyPtr;

/* Network allocation vector time. This time is absolute in regard to the simulation. */
double  \navTime;

/* The size of the current aggregatted fragments that are ready to be transmitted in the
fragmentation list. If it is for the first time then correspond to the packet size. */
OpT_Packet_Size        \currentPacketSize;

/* The arrival time of the packet that is currently handled by the DCF. */
double  \receiveTime;

/* The type of frame that the station needs to transmit next */
UanT_Mac_Frame_Type    \frameTypeToSend;

/* Set the expected frame type needed in response to the transmitted frame
*/
UanT_Mac_Frame_Type    \expectedFrameType;

double  \rxEndTime;

/* Intrpt type is stored in this variable */
int     \interruptType;

/* Enumerated intrpt code for interrupts */
```

```
UwnT_Mac_Intrpt_Code   \interruptCode;

/* Pool memory handle used to allocate memory for the data received from the higher layer
and inserted in the queue */
Pmohandle      \highLayerPMH;

/* This variable maintains the maximum size of the higher layer data buffer as specified
by the user.    */
OpT_Packet_Size        \highLayerListMaxSize;

/* Maintaining total size of the packets in higher layer queue. */
OpT_Packet_Size        \highLayerListTotalSize;

/* Higher layer data arrival queue or list */
List*   \highLayerListPtr;

/* Monitor queue size as the packets arrive from higher layer. */
Stathandle     \highLayerPacketRcvd;

/* The channel's error rate that will affect the effective data rate */
double  \errorRate;

/* The propagation speed of the signal */
double  \propagationSpeed;

/* Time that all nodes in the channel need to sense the medium free before transmmiting
*/
double  \difsDuration;

/* Minimum contention window size for selecting backoff slots */
int     \minContentionWindow;

/* Maximum contention window size for selecting backoff slots */
int     \maxContentionWindow;

/* The recipient MAC address */
int     \destinationAddress;

/* The data rate of the single outbound channel */
double  \outboundChannelDataRate;

/* SIFS - Short Inter Frame Spacing. The time that the node needs to wait between frame
transmission, or as a direct response to a frame */
double  \sifsDuration;

/* Keeps track of the current simulation time at each interrupt */
double  \currentTime;

/* Last simulation time when the receiver became idle */
double  \receiverIdleTime;

/* Event handle that keeps track of the self interrupt due to Deference. */
Evhandle       \deferenceEVH;

/* EIFS duration which is used when the station receives an erroneous frame */
double  \eifsDuration;

/* Receiver channel state information. */
UwnT_Rx_State_Info*    \rxStateInfoPtr;

/* Event handle that keeps track of the self interrupt due to frame timeout when the
station is waiting for a response.     */
Evhandle       \frameTimeoutEVH;

/* This frame keeps track of the last transmitted frame that needs a frame response (like
Cts or Ack). This is actually used when there is a need for retransmission.
*/
UanT_Mac_Frame_Type    \lastFrameTxType;

/* Extracting remote station's address from the received packet */
int     \remoteAddress;
```

```
/* Holds the retry limit defined by the related attribute */
int     \retryLimit;

/* Flag for duplicate entry  Keeps track that whether the receive frame was a duplicate
frame or not. This information is transmitted in ACK frame. */
int     \duplicateEntry;

/* Maximum time after the initial reception of the fragmented MSDU after which further
attempts to reassemble the MSDU will be terminated. */
double  \maxReceiveLifetime;

/* Maximum backoff value for picking random backoff interval */
int     \maxBackoff;

/* Storing total backoff time when backoff duration is set */
double  \interruptTime;

/* Read the Slot duration from the model attributes */
double  \slotDuration;

/* Event handle that keeps track of the self interrupt due to backoff. */
Evhandle        \backoffElapsedEVH;

/* Counter to determine packet sequence number for each transmitted packet */
int     \packetSequenceNr;

/* Size of the last data fragment */
OpT_Packet_Size         \remainderSize;

/* Store packet id of the data packet in service. */
OpT_Packet_Id  \packetInService;

/* The arrival time of the packet that is currently handled by the MAC */
double  \receiveTimeMAC;

/* The variable that holds the network mode */
int     \networkMode;

/* The object ID of the surrounding network */
Objid   \myNetworkObjectID;

/* The name of the destination node */
char    \destinationNodeName[25];

/* The handle for the error rate distribution */
Distribution * \errorRateDistribution;

/* Variable that holds the attribute that defines the type of this node */
int     \typeOfNode;

/* The time until the receiver will be occuppied */
double  \receiverEndTime;

/* Variable that keeps track of the number of ON and OFF interrupts received from the
receiver. With a ON it adds one. With a OFF it takes one. The receiver is idle when this
variable is zero. */
int     \receiverInterruptsSemafor;

char    \myName[10];

int     \dataPacketIDRcvd;

/* Holds the simulation time when the node receives a Cts as a response to a previous Rts
*/
double  \timeCtsReceived;

/* Holds the simulation when the node sends a Rts */
double  \timeRtsSend;

/* Handle for the pool memory to allocate memory for the the data structure elements that
```

```
hold the fragments of a packet to send */
Pmohandle      \fragmentationPMH;

/* This contains the data structure elements that contain the fragments of a packet to
send */
List *  \fragmentationList;

/* The fragments received of the packet being transmitted */
int     \receivedFragments;

double  \previousNav;

int     \currentIndexDefragmentationBuffer;

Stathandle     \eteDelayBackgroundTraffic;

Stathandle     \eteDelayNonPeriodicTraffic;

double  \eteDelayAllTrafficByNode[MAX_NUMBER_NODES];

double  \allTrafficReceivedByNode[MAX_NUMBER_NODES];

int     \allPacketsReceivedByNode[MAX_NUMBER_NODES];

double  \constTrafficReceivedByNode[MAX_NUMBER_NODES];

double  \eventTrafficReceivedByNode[MAX_NUMBER_NODES];

double  \eteDelayConstTrafficByNode[MAX_NUMBER_NODES];

double  \eteDelayEventTrafficByNode[MAX_NUMBER_NODES];

int     \constPacketsReceivedByNode[MAX_NUMBER_NODES];

int     \eventPacketsReceivedByNode[MAX_NUMBER_NODES];

================================================================================
                           Temporary Variable Block
================================================================================
Boolean                                pre_rx_status;
Boolean                                bad_packet_rcvd = OPC_FALSE;

UanT_Mac_Fragmentation_List_Element* sentFragment;

int idx;
double timeout;

================================================================================
                                 Function Block
================================================================================
/*
################################################################################################
################################################################################################
##
This function drops the higher layer packets or packets received by the relay nodes that
needed to be forwarded but cannot be accepted because of full buffer. It also writes an
appropriate log message to report the rejection unless the same log message is already
written before */

static void high_layer_packet_drop (Packet* highLayerCurrPkt)
{
       double  currDataSize;

       FIN (high_layer_packet_drop (highLayerCurrPkt));

       currDataSize = (double) op_pk_total_size_get (highLayerCurrPkt);

       /* Write an appropriate simulation log message unless the same message is written
          before */
       if (dropPacketEntryLogFlag == OPC_FALSE)
       {
```

```
                // Writing log message for dropped packets
                op_prg_log_entry_write (dropPacketLogHandle,
                        "SYMPTOM(S):\n"
                        " The network MAC layer discarded some packets due to\n "
                    " insufficient buffer capacity. \n"
                        "\n"
                    " This may lead to: \n"
                    " - application data loss.\n"
                    " - higher layer packet retransmission.\n"
                    "\n"
                    " REMEDIAL ACTION(S): \n"
                    " 1. Reduce network load. \n"
                    " 2. Increase network data rate. \n"
                    " 3. Increase buffer capacity\n");
                        dropPacketEntryLogFlag = OPC_TRUE;
        }

        // Destroy the dropped packet
        op_pk_destroy (highLayerCurrPkt);

        // Report stat when data packet is dropped due to overflow buffer
        op_stat_write (dropPacketHandle, 1.0);
//      op_stat_write (dropPacketHandle, 0.0);

        // Report stat when data packet is dropped due to overflow buffer
        op_stat_write (dropPacketHandleInBits, (double) (currDataSize));
        op_stat_write (dropPacketHandleInBits, 0.0);
        op_stat_write (globalDroppedDataHandle, (double) (currDataSize));
        op_stat_write (globalDroppedDataHandle, 0.0);

        // Update the dropped packets statictics
        packetsDroppedByNode[myAddress]++;

        FOUT;
}


/*****************************************************************************************
##
*****************************************************************************************
##
Enqueuing data packet for transmission */

static void high_layer_packet_enqueue (Packet* highLayerCurrPkt, int stream)
{
        UanT_High_Layer_List_Elem*    highLayerPtr;
        double                                          currDataSize;

        FIN (high_layer_packet_enqueue (highLayerCurrPkt, stream));

        // Allocating pool memory to the higher layer data structure type
        highLayerPtr = (UanT_High_Layer_List_Elem *) op_prg_pmo_alloc (highLayerPMH);

        /* Generate error message and abort simulation if no memory left for data received
           from higher layer */
        if (highLayerPtr == OPC_NIL)
                mac_error ("No more memory left to assign for data received from higher
layer",
                                OPC_NIL, OPC_NIL);

        // Updating higher layer data structure fields
        highLayerPtr->time_rcvd = op_sim_time();
        highLayerPtr->typeOfTraffic = stream;
        highLayerPtr->pkptr = highLayerCurrPkt;

        printf("\n#%s# - Just before enqueueing the packet\n", myName);

        // Insert a packet to the list
        op_prg_list_insert (highLayerListPtr, highLayerPtr, OPC_LISTPOS_TAIL);

        printf("\n#%s# - After enqueueing the packet\n", myName);
```

```
        // Enable the flag indicating that there is a data frame to transmit
        flags->packet_to_send = OPC_TRUE;

        // Report stat when outbound data packet is received
        op_stat_write (packetLoadHandle, 1.0);

        // Report stat in bits when outbound data packet is received
        currDataSize = (double) op_pk_total_size_get (highLayerCurrPkt);
        op_stat_write (bitsLoadHandle, currDataSize);
    op_stat_write (bitsLoadHandle, 0.0);

        // Update the global statistics as well
        op_stat_write (globalLoadHandle, currDataSize);
    op_stat_write (globalLoadHandle, 0.0);

        // Update the queue size statistic
        op_stat_write (highLayerPacketRcvd, (double) op_prg_list_size (highLayerListPtr));


        FOUT;
}


/*##############################################################################################
##
##############################################################################################
##
Fragment the first packet of the higher layer queue and put the fragments in the
fragmentation queue from where they will be dequeued to send. The fragments are in
compliance with the size defined in the model attributes, namely the data frame payload
size. When it comes the time to be transmited they only need to be dequeued. The
data structure elements in the fragmentation queue also keep track of whether the
fragment was acknowledged by the receiving station, and only after that they will be
destroyed */

static void build_packet_fragments_list ()
{
        UanT_Mac_Fragmentation_List_Element* packetFragment;
        UanT_High_Layer_List_Elem*    highLayerPtr;
        OpT_Sar_Size bufferSize;
        Sbhandle fragmentationBuffer;
        int numberOfFragments;
        int fragmentNumber;
        double pkt_tx_time;

        int idx;

        FIN (build_packet_fragments_list());

        fragmentationBuffer = op_sar_buf_create (OPC_SAR_BUF_TYPE_SEGMENT,
OPC_SAR_BUF_OPT_PK_BNDRY);

        // Remove packet from higher layer queue
        highLayerPtr = (UanT_High_Layer_List_Elem*) op_prg_list_remove(highLayerListPtr,
0);

        // Update the higher layer queue size statistic
        op_stat_write (highLayerPacketRcvd, (double) op_prg_list_size (highLayerListPtr));

        printf("\n#%s# - Inside build_fragments_list()", myName);

        // Updating the total packet size of the higher layer buffer
        currentPacketSize = op_pk_total_size_get (highLayerPtr->pkptr);
        highLayerListTotalSize = highLayerListTotalSize - currentPacketSize;

        // Packet seq number modulo 4096 counter
        packetSequenceNr = (packetSequenceNr + 1) % 4096;

        // Storing Data packet id for debugging purposes
        packetInService = op_pk_id (highLayerPtr->pkptr);
```

```
	// Store the arrival time of the packet
	receiveTimeMAC = highLayerPtr->time_rcvd;

	/* Computing packet duration in the queue in seconds and reporting it to the
	   statistics */
	// DO I REALLY NEED THIS. CHECK ALSO receiveTimeMAC
	pkt_tx_time = currentTime - receiveTimeMAC;

	printf ("\n#%s# - Data packet " OPC_PACKET_ID_FMT " is removed from higher layer
buffer", myName, packetInService);
	printf ("\n#%s# -The queueing delay for data packet " OPC_PACKET_ID_FMT " is %fs",
myName, packetInService, pkt_tx_time);
	printf("\n#%s# Putting the packet into the fragmentation buffer to send. Packet
ID: %i\n", myName, (int)op_pk_id(highLayerPtr->pkptr));

	// Packet needs to be fragmented if it is greater than the data frame payload size
	if (currentPacketSize > sizeDataFramePayload)
	{
		/* Determine the number of fragments for the packet and the size of the
last
		fragment */
		numberOfFragments = (int) (currentPacketSize / sizeDataFramePayload);

		if ((currentPacketSize - (numberOfFragments * sizeDataFramePayload))
			!= 0)
			numberOfFragments++;
	}
	else
		numberOfFragments = 1;

	// Packet fragment number is initialized
	fragmentNumber = 0;

	printf("\n#%s# Putting the packet into the fragmentation buffer to send. Packet
ID: %i\n", myName, (int)op_pk_id(highLayerPtr->pkptr));

	// Insert packet to fragmentation buffer
	op_sar_segbuf_pk_insert (fragmentationBuffer, highLayerPtr->pkptr, 0);

	printf("\n#%s# - The size of the fragmentation buffer = %i", myName,
op_sar_buf_size (fragmentationBuffer));


	for (idx = 0; idx < numberOfFragments; idx++)
	{
		// Allocating pool memory for the fragmentation data structure type
		packetFragment = (UanT_Mac_Fragmentation_List_Element*)
					op_prg_pmo_alloc (fragmentationPMH);

		// Set fields in the data structure
		packetFragment->fragmentNumber = fragmentNumber;
		packetFragment->transmitted = OPC_FALSE;
		packetFragment->typeOfTraffic = highLayerPtr->typeOfTraffic;

		// Set buffer size
		bufferSize = op_sar_buf_size (fragmentationBuffer);

		if (bufferSize > sizeDataFramePayload)
		{
			packetFragment->fragmentSize = sizeDataFramePayload;
			packetFragment->fragment = op_sar_srcbuf_seg_remove
					(fragmentationBuffer,
sizeDataFramePayload);
		}

		else
		{
			packetFragment->fragmentSize = bufferSize;
			packetFragment->fragment = op_sar_srcbuf_seg_remove
					(fragmentationBuffer, bufferSize);
```

```
		}

		packetFragment->numberOfFragments = numberOfFragments;

		op_prg_list_insert (fragmentationList, packetFragment, OPC_LISTPOS_TAIL);

		fragmentNumber++;
	}

	// Set the flag stating that we have fragments ready to send
	flags->fragments_to_send = OPC_TRUE;

	// If it is the case, update the flag of packets to send
	if (op_prg_list_size (highLayerListPtr) == 0 &&
		op_sar_buf_size (fragmentationBuffer) == 0)
		flags->packet_to_send = OPC_FALSE;

	// Free up allocated memory for the fragmentation buffer
	op_sar_buf_destroy (fragmentationBuffer);

	// Free up allocated memory for the data packet removed from the higher layer
queue
	op_prg_mem_free (highLayerPtr);

	FOUT;

}


/* ########################################################################## */
/* ########################################################################## */
/* ########################################################################## */
/****** Error handling procedure ******/
static void mac_error (const char* msg1, const char* msg2, const char* msg3)
	{

	/** Terminates simulation with an error message.     **/
	FIN (mac_error (msg1, msg2, msg3));

	op_sim_end ("Error in UAN process:", msg1, msg2, msg3);

	FOUT;
	}
/*

                               */
/*
##############################################################################################
#########################
##############################################################################################
#########################
This routine schedules self interrupt for deference to avoid collision and also deference
to observe interframe
gap between the frame transmission.
*/
static void schedule_deference ()
{
	double interrupt;

	FIN (schedule_deference ());

	/* Check the status of the receiver. If it is busy, exit the function, since we
will schedule the end of the
	   deference when it becomes idle. */
	if (flags->receiver_busy == OPC_TRUE)
		FOUT;

	// Extracting current time at each interrupt
	currentTime = op_sim_time ();

	// Adjust the NAV if necessary
```

```
        if (navTime < receiverIdleTime)
                navTime = receiverIdleTime;

        /* If the frame type to send is none, our next frame type may be an Rts, and we
may
           need to resume an interrupted backoff period. Set the appropriate state
variables
           in order to schedule the correct deference */

                /* After a relay node interrupts its backoff period in order to answer to
a
                   sensor node request, it needs to perform backoff after the completion
of that
                exchange and before the relay node attempt to initiate an exchange with its
                destination node */

        interrupt = (navTime - (receiverIdleTime + difsDuration)) > PRECISION_RECOVERY ?
navTime :
                                                (receiverIdleTime + difsDuration);


        // If we need to perform backoff we set the end of deference to navTime
        if (flags->backoff_required == OPC_TRUE && frameTypeToSend != UanE_Cts &&
frameTypeToSend != UanE_Ack)
        {
        printf("\n#%s# - Schedule deference - Backoff required", myName);

                flags->perform_backoff = OPC_TRUE;

                if (frameTypeToSend == UanE_Rts_Ime)
                        frameTypeToSend = UanE_Rts;

                if (networkMode == UanE_Aloha_Alike)
                        deferenceEVH = op_intrpt_schedule_self (currentTime,
UwnE_Deference_Off);
                else
                        deferenceEVH = op_intrpt_schedule_self (navTime,
UwnE_Deference_Off);

        }

        else if (networkMode == UanE_Aloha_Alike)
        {
                if (frameTypeToSend == UanE_Rts)
                        deferenceEVH = op_intrpt_schedule_self (currentTime,
UwnE_Deference_Off);

                else
                {
                        if (frameTypeToSend == UanE_Rts_Ime)
                                frameTypeToSend = UanE_Rts;

                        deferenceEVH = op_intrpt_schedule_self (currentTime + sifsDuration,
UwnE_Deference_Off);
                }
        }

        // Schedule a self interrupt in order for schedule the deference after the end of
navTime
        else if (frameTypeToSend == UanE_Rts_Ime)
                {
                frameTypeToSend = UanE_Rts;
                deferenceEVH = op_intrpt_schedule_self (interrupt, UwnE_Deference_Off);
                }

        else if (frameTypeToSend == UanE_Rts)
        {
                if (MEDIUM_IS_IDLE)
                {
                        deferenceEVH = op_intrpt_schedule_self (currentTime,
UwnE_Deference_Off);
```

```
		}
		else
		{
			flags->perform_backoff = OPC_TRUE;
			flags->backoff_required = OPC_TRUE;

			if (navTime < currentTime)
				navTime = currentTime;

			deferenceEVH = op_intrpt_schedule_self (navTime,
UwnE_Deference_Off);
		}
	}

	/* Station needs to wait SIFS duration before responding to any frame. Also, if
Rts/Cts is enabled then the
	   station needs to wait for SIFS duration after acquiring the channel using
Rts/Cts exchange. If more
	   fragments to send then wait for SIFS duration and transmit. */
	else if (frameTypeToSend != UanE_None)
	{

		deferenceEVH = op_intrpt_schedule_self (currentTime + sifsDuration,
UwnE_Deference_Off);

	}


	else
	{
	mac_error ("WRONG PLACE IN DEFERENCE????", OPC_NIL, OPC_NIL);
	}

	/* Reset the updated NAV flag, since as of now we scheduled a new "end of
deference" interrupt after the last
	   update */

	flags->nav_updated = OPC_FALSE;

	FOUT;
}


/*
##############################################################################################
##
##############################################################################################
##
This function queues the packet as it arrives from higher layer
*/
static void higher_layer_data_arrival (void)
{
	Packet*				currentPacket;
	OpT_Packet_Size			dataSize, fragSize;

	FIN (higher_layer_data_arrival (void));

	// Get packet from the incoming stream from higher layer
	currentPacket = op_pk_get (op_intrpt_strm ());

	printf("\n#%s# - Packet from generator: %i\n", myName, (int)op_pk_id
(currentPacket));

	/* Get the size of the packet arrived from higher layer.		*/
	dataSize = op_pk_total_size_get (currentPacket);

	/* If packet size is greater than the data frame payload size defined, then the
	   fragments will not be greater than the data frame payload size defined */
	if (dataSize > sizeDataFramePayload)
		fragSize = sizeDataFramePayload;
	else
```

```
                fragSize = dataSize;

        // Sum of the accumulated bits and packets generated by the node
        if (op_intrpt_strm() == UanE_Background_Traffic)
                {
        bitsGeneratedByNode[myAddress] += dataSize;
        packetsGeneratedByNode[myAddress]++;
        }

        /* Destroy packet if it has size zero. Also, if the size of the higher layer queue
           will exceed its maximum after the insertion of this packet, then discard the
           arrived packet. The higher layer is responsible for the retransmission of this
           packet */
        if (dataSize == 0 || (highLayerListTotalSize + dataSize > highLayerListMaxSize))
        {
                // Drop the higher layer packet
                high_layer_packet_drop (currentPacket);
                FOUT;
        }


        /* Stamp the packet with the current time. This information will remain unchanged
even
           if the packet is copied for retransmissions, and eventually it will be used by
the
           destination MAC to compute the end-to-end delay */
        op_pk_stamp (currentPacket);

        printf("\n#%s# Stamp packet from higher layer %f\n", myName,
op_pk_stamp_time_get(currentPacket));

        /* Maintaining total packet size of the packets in the higher layer queue */
        highLayerListTotalSize = highLayerListTotalSize + dataSize;

        /* Insert the arrived packet in higher layer queue.                          */
    high_layer_packet_enqueue (currentPacket, op_intrpt_strm());

        // Update packets generated statistics
        packetsInQueueByNode[myAddress]++;


        FOUT;
}
/*
##############################################################################################
##
##############################################################################################
##
This routine handles the appropriate processing need for each type of remote interrupt.
The type of interrupts are: stream interrupts (from lower and higher layers), stat
interrupts (from receiver and transmitter)
*/
static void interrupts_process (void)
{
        int stream = INT_ZERO;

        FIN (interrupts_process (void));

        // Determine the current simulation time
        currentTime = op_sim_time ();

        // Determine interrupt type and code to divide treatment later
        interruptType = op_intrpt_type ();

        // Determine if this is the end of simulation and record final stats
        if (op_intrpt_code() == OPC_INTRPT_ENDSIM && typeOfNode == UanE_Gateway_Node)
        {
                printf("\n####End of simulation####");
                record_final_stats();
                FOUT;
        }
```

```
	// Interrupts above 50 are physical interrupts (model defined)
	if (op_intrpt_code() < 50)
		interruptCode = (UwnT_Mac_Intrpt_Code) op_intrpt_code ();
	else
		interruptCode = (UwnT_Physical_Interrupts) op_intrpt_code();

	/* Stream interrupts are either arrivals from the higher layer, or from the
physical
	   layer */
	if (interruptType == OPC_INTRPT_STRM)
	{
		stream = op_intrpt_strm();

		switch (stream)
		{
			// If the event arrived from higher layer then queue the packet
			case INPUT_STREAM_FROM_UPPER_LAYER_BACKGROUND:
			case INPUT_STREAM_FROM_UPPER_LAYER_NON_PERIODIC:
			{
				// Process stream interrupt received from higher layer
				printf("\n#%s# nCalling higher_layer_data_arrival()\n",
myName);
				higher_layer_data_arrival();
				break;
			}

			/* If the event was an arrival from the physical layer, accept the
packet and
			   decapsulate it. Later it will be tested for collision and errors
*/
			case INPUT_STREAM_FROM_PHYSICAL_LAYER:
			{
				printf("\n#%s# Calling physical_layer_data_arrival() -
Receiver semafor: %i\n", myName, receiverInterruptsSemafor);
				physical_layer_data_arrival();
				break;
			}

			default:
			{
				mac_error ("Unexpected stream interrupt encountered.",
OPC_NIL, OPC_NIL);
				break;
			}

		} // end switch stream

	}

	/* Handle interrupt received from the receiver. This is the start of the reception
of
	   a new frame */
	else if (interruptType == OPC_INTRPT_REMOTE && interruptCode == UwnE_Receiver_On)
	{
		printf("\n#%s# Receiver ON\n", myName);
		// If the receiver was already busy, set the flags collision on
		if (flags->receiver_busy)
			flags->collision = OPC_TRUE;

		// Set the receiver status as busy and increase the receiver semafor
		flags->receiver_busy = OPC_TRUE;
		receiverInterruptsSemafor++;
	}

	/* Else a packet reception is complete. Check whether the receiver became
available
	   while it was busy*/
	else if (interruptType == OPC_INTRPT_REMOTE && interruptCode == UwnE_Receiver_Off)
	{
		printf("\n#%s# Receiver OFF\n", myName);
```

```
		// Decrease the receiver semafor
		receiverInterruptsSemafor--;

		// If the receiver semafor is equal to zero, it means the receiver is idle
		if (receiverInterruptsSemafor == 0)
		{
			// Update de collision flag
			flags->collision = OPC_FALSE;

			// Update the receiver status
			flags->receiver_busy = OPC_FALSE;

			// Update the medium idle time with current time
			receiverIdleTime = currentTime;

		}

	}

	// Check whether we need to set the the frame type to send flag to Rts
	if ((flags->packet_to_send == OPC_TRUE || flags->fragments_to_send == OPC_TRUE) &&
		flags->rts_sent == OPC_FALSE && frameTypeToSend == UanE_None &&
expectedFrameType == UanE_None)
	{
		frameTypeToSend = UanE_Rts;
	}


	FOUT;
}


/*##############################################################################################
##
##############################################################################################
##
Process the frame received from the lower layer. This routine decapsulate the frame and
set
appropriate flags if the station needs to generate a response to the received frame
*/
static void physical_layer_data_arrival (void)
{
	UanT_Frame_Fields*
	rcvdDummyHeader;
	Packet*
	rcvd_frame_ptr;
	Packet*
	seg_pkptr;
	Boolean
	data_pkt_received        = OPC_FALSE;
	Boolean
	disable_signal_extension = OPC_FALSE;
	double
	rcvd_pk_size, rx_start_time;
	double
	previous_nav;
	double mac_delay;
	int idx, allFragmentsTransmitted;
	UanT_Mac_Fragmentation_List_Element* sentFragment;

	FIN (physical_layer_data_arrival (void));


	//  Access received packet from the physical layer stream
	rcvd_frame_ptr = op_pk_get (INPUT_STREAM_FROM_PHYSICAL_LAYER);

	// Getting the dummy header
	op_pk_fd_get (rcvd_frame_ptr, 0, &rcvdDummyHeader);

	/* If the packet is received while the station is in transmission, or if the
packet is
```

```
           collided with another packet received or if it is an error frame, then the
packet
           will not be processed and if needed the station will retransmit the packet */
        if ((flags->rcvd_bad_packet == OPC_TRUE) || (errorFrame() == OPC_TRUE) ||
              (flags->collision == OPC_TRUE))
        {
              printf("\n#%s# - Expecting none. Received bad packet from %i", myName,
rcvdDummyHeader->senderAddress);

              if (flags->collision == OPC_TRUE)
                     framesCollidedByNode[myAddress]++;


              // Reset the bad packet receive flag for subsequent receptions
              flags->rcvd_bad_packet = OPC_FALSE;

              // Set backoff required
              flags->backoff_required = OPC_TRUE;

              // Destroy the bad packet
              op_pk_destroy (rcvd_frame_ptr);

              // Break the routine as no further processing is needed
              FOUT;
        }

        /* Store the greater of current time and current value of NAV before processing
the
           packet to be used to update the channel reservation statistic if NAV is updated
due
           to the received packet. */
        previous_nav = (navTime > currentTime) ? navTime : currentTime;

        // Compute the values that will be used while updating the received traffic
statistics.
        */
        rcvd_pk_size  = (double) op_pk_total_size_get (rcvd_frame_ptr);
        rx_start_time = currentTime - rcvd_pk_size / rcvdDummyHeader->dataRate;

        /* Split the logic in accordance with the current state of the node
           that is, in accordance with the expected frame type */
        switch (expectedFrameType)
        {

              // The node is expecting none
              case (UanE_None):
              {

                     // Split the logic according to the frame type received
                     switch (rcvdDummyHeader->frameType)
                     {

                            // Expecting none and received an Rts
                            case (UanE_Rts):
                            {
                                   /*Update received control traffic statistics. Write
the appropriate values for
                                     start and end of the reception */
                                   op_stat_write_t (ctrlTrafficRcvdHandleInBits,
rcvd_pk_size, rx_start_time);
                                   op_stat_write   (ctrlTrafficRcvdHandleInBits, 0.0);
                                   op_stat_write_t (ctrlTrafficRcvdHandle,       1.0,
rx_start_time);
                                   op_stat_write   (ctrlTrafficRcvdHandle,       0.0);

                                   /* We will respond to the Rts with a Cts only if a)
the     Rts is destined for us,
                                      and b) our NAV duration is not larger than
current simulation time */
                                   if (rcvdDummyHeader->receiverAddress == myAddress &&
currentTime >= navTime)
```

```
                                        {
                                                // Retrieve the sender address because it
may be needed later
                                                remoteAddress = rcvdDummyHeader-
>senderAddress;

                                                // Set the frame response field to Cts
                                                frameTypeToSend = UanE_Cts;

                                                expectedFrameType = UanE_None_Transit;

                                                // Delay a possible backoff period
                                                if (flags->perform_backoff == OPC_TRUE)
                                                        flags->perform_backoff = OPC_FALSE;

                                                // Printing out information to ODB
                                                printf ("\n#%s# - Rts is received and Cts
will be transmitted",myName);
                                                printf ("\n#%s# - Remote address defined =
%i",myName, remoteAddress);
                                        }

                                        else
                                                // Printing out information to ODB
                                                printf ("\n#%s# - Rts is received and
discarded", myName);

                                        // Update NAV
                                        navTime = rcvdDummyHeader->senderTime +
rcvdDummyHeader->reservedDuration;
                                        flags->nav_updated = OPC_TRUE;

                                        break;

                                } // End case for expecting none and received an Rts

                                // Expecting none and received a Cts
                                case (UanE_Cts):
                                {
                                        /* Update received control traffic statistics. Write
the appropriate values for
                                           start and end of the reception */
                                        op_stat_write_t (ctrlTrafficRcvdHandleInBits,
rcvd_pk_size, rx_start_time);
                                        op_stat_write (ctrlTrafficRcvdHandleInBits, 0.0);
                                        op_stat_write_t (ctrlTrafficRcvdHandle, 1.0,
rx_start_time);
                                        op_stat_write (ctrlTrafficRcvdHandle, 0.0);

                                        printf("\n#%s# Expecting none. Received Cts from
%i", myName, rcvdDummyHeader->senderAddress);

                                        if (rcvdDummyHeader->receiverAddress != myAddress)
                                        {
                                                // Update the NAV time and set the updated
flag
                                                navTime = rcvdDummyHeader->senderTime +
rcvdDummyHeader->reservedDuration;
                                                flags->nav_updated = OPC_TRUE;
                                        }

                                        break;

                                } // End case for expecting none and received a Cts

                                // Expecting none and received a data frame
                                case (UanE_Data):
                                {

                                        /* Update received data traffic statistics. Write
the appropriate values for start
```

```
                                        and end of the reception */
                                    op_stat_write_t (dataTrafficRcvdHandleInBits,
rcvd_pk_size, rx_start_time);
                                    op_stat_write (dataTrafficRcvdHandleInBits, 0.0);
                                    op_stat_write_t (dataTrafficRcvdHandle, 1.0,
rx_start_time);
                                    op_stat_write (dataTrafficRcvdHandle, 0.0);

                                    printf("\n#%s# Expecting none. Received data packet
" OPC_PACKET_ID_FMT " from %i", myName, rcvdDummyHeader->dataPacketID,rcvdDummyHeader-
>senderAddress);

                                    // Update NAV duration and set the updated flag
                                    navTime = rcvdDummyHeader->senderTime +
rcvdDummyHeader->reservedDuration;
                                    flags->nav_updated = OPC_TRUE;

                                    /* If we are in Aloha alike mode, and data is
                                       for this node, then this is a legitimate data
frame */
                                    if (networkMode == UanE_Aloha_Alike &&
rcvdDummyHeader->receiverAddress == myAddress)
                                    {
                                            // Delay a possible backoff period
                                            if (flags->perform_backoff == OPC_TRUE)
                                                    flags->perform_backoff = OPC_FALSE;

                                            // Retrieve the fragment
                                            op_pk_fd_get (rcvd_frame_ptr, 1,
&seg_pkptr);

                                            // Call the processing routine
                                            data_process (seg_pkptr, rcvdDummyHeader);
                                    }

                                    break;

                            } // End case for expecting none and received a data frame

                            // Expecting none and received an Ack
                            case (UanE_Ack):
                            {
                                    /* Update received control traffic statistics. Write
the appropriate values for
                                       start and end of the reception */
                                    op_stat_write_t (ctrlTrafficRcvdHandleInBits,
rcvd_pk_size, rx_start_time);
                                    op_stat_write (ctrlTrafficRcvdHandleInBits, 0.0);
                                    op_stat_write_t (ctrlTrafficRcvdHandle, 1.0,
rx_start_time);
                                    op_stat_write (ctrlTrafficRcvdHandle, 0.0);

                                    printf("\n#%s# Expecting none. Received Ack from
%i", myName, rcvdDummyHeader->senderAddress);

                                    // Update NAV duration and set the updated flag
                                    navTime = rcvdDummyHeader->senderTime +
rcvdDummyHeader->reservedDuration;
                                    flags->nav_updated = OPC_TRUE;

                                    break;

                            } // End case for expecting none and received a data frame

                            default:
                            {
                                    // Unknown frame type so declare error
                                    mac_error ("Unexpected frame type received.",
OPC_NIL, OPC_NIL);
                            }
```

```
                        } // End switch that splits the flow according the type of the
received frame

                        break;

                } // End case expectedFrameType = UanE_None


                case (UanE_None_Transit):
                {
                        // Do nothing because the node is currently communicating with some
other node
                        printf("\n#%s# - Received a frame. The node is currently in an
exchange process. Do nothing", myName);
                        break;
                }

                // The node is expecting a Cts
                case (UanE_Cts):
                {

                        // Depending on the received frame type
                        switch (rcvdDummyHeader->frameType)
                        {

                                // Backoff only if the received RTS is from the destination
node
                                case (UanE_Rts):
                                {
                                        updateControlTrafficStats (rcvd_pk_size,
rx_start_time);

                                        // If this frame is from node's destination address
then there exists a problem
                                        if (rcvdDummyHeader->senderAddress ==
destinationAddress)
                                        {
                                                // Update NAV and set the updated flag
                                                navTime = rcvdDummyHeader->senderTime +
rcvdDummyHeader->reservedDuration;
                                                flags->nav_updated;

                                                // Set the appropriate flags to backoff and
restart the process
                                                flags->rts_sent = OPC_FALSE;
                                                expectedFrameType = UanE_None;
                                                flags->backoff_required = OPC_TRUE;


                                                // Schedule a resume timeout event
                                                op_intrpt_schedule_self (op_sim_time(),
UanE_Resume_Timeout);

                                                /* Increment the retransmission counter and
check whether further
                                                   retries are possible or the packet, or
its remains, needs to be discarded */
                                                retryCount++;
                                                frame_discard ();
                                        }

                                        break;

                                } // End case where expecting a Cts and received a Rts

                                // Received a CTS
                                case (UanE_Cts):
                                {
                                        updateControlTrafficStats (rcvd_pk_size,
rx_start_time);

                                        /* Check whether the frame is destined for this node
```

```
then set
                                        appropriate indicators. Otherwise, just update
the NAv if
                                        the received NAV is greater */
                                    if (rcvdDummyHeader->receiverAddress == myAddress)
                                    {
                                            /* The receipt of Cts frame indicates that
Rts is successfully transmitted and
                                               the station can now respond with Data
frame */
                                            frameTypeToSend = UanE_Data;

                                            // Set the flag indicating that Rts is
successfully transmitted
                                            flags->rts_sent = OPC_TRUE;

                                            // Set the time that the Cts is received for
calculating node's distance
                                            timeCtsReceived = currentTime;

                                            // Set expected frame flag
                                            expectedFrameType = UanE_None_Transit;

                                            // Set remote address for future use
                                            remoteAddress = rcvdDummyHeader-
>senderAddress;

                                            // Schedule a resume timeout event
                                            op_intrpt_schedule_self (op_sim_time(),
UanE_Resume_Timeout);

                                            /* If we are accessing the media then this
the end of media access
                                               duration. This statistics reports, not
only the duration for the
                                               first attempt, but also attempts made for
possible retransmissions */
                                            mac_delay = currentTime - receiveTimeMAC;
                                            op_stat_write (mediaAccessDelay, mac_delay);
                                            op_stat_write (globalMACdelayHandle,
mac_delay);

                                            // op_stat_write (retransHandle, (double)
(retryCount * 1.0));

                                            // Printing out information to ODB
                                            printf ("\n#%s# - Cts is received for Data
packet "
                                                    OPC_PACKET_ID_FMT, myName,
packetInService);
                                    }

                                    // If not for this node
                                    else
                                    {
                                            // Printing out information to ODB
                                            printf ("\n#%s# - Cts is received and
discarded.", myName);

                                            // Set the appropriate flags to backoff and
restart the process
                                            flags->rts_sent = OPC_FALSE;
                                            expectedFrameType = UanE_None;

                                            // Schedule a resume timeout event
                                            op_intrpt_schedule_self (op_sim_time(),
UanE_Resume_Timeout);

                                            flags->backoff_required = OPC_TRUE;

                                            /* Increment the retransmission counter and
```

```
check whether further
                                                retries are possible or the packet, or
its remains, needs to be discarded */
                                        retryCount++;
                                        frame_discard ();

                                        // Update NAV
                                        navTime = rcvdDummyHeader->senderTime +
rcvdDummyHeader->reservedDuration;
                                        flags->nav_updated;

                                        // Set the flag that indicates updated NAV
value
                                        flags->nav_updated = OPC_TRUE;

                                }

                                break;

                        } // End case the expected and received frame type is a Cts

                        // Do not disturb a ongoing communication
                        case (UanE_Data):
                        {
                                updateDataTrafficStats(rcvd_pk_size, rx_start_time);

                                // Update NAV and set the updated flag
                                navTime = rcvdDummyHeader->senderTime +
rcvdDummyHeader->reservedDuration;
                                flags->nav_updated;

                                // Set the appropriate flags to backoff and restart
the process
                                flags->rts_sent = OPC_FALSE;
                                expectedFrameType = UanE_None;

                                // Schedule a resume timeout event
                                op_intrpt_schedule_self (op_sim_time(),
UanE_Resume_Timeout);

                                flags->backoff_required = OPC_TRUE;

                                /* Increment the retransmission counter and check
whether further
                                   retries are possible or the packet, or its
remains, needs to be discarded */
                                retryCount++;
                                frame_discard ();

                                break;

                        } // End the case where the expected frame type is a Cts
and the received is a data frame


                        // Do nothing. It is possible that the RTS sent get through
                        case (UanE_Ack):
                        {
                                updateControlTrafficStats (rcvd_pk_size,
rx_start_time);
                                break;
                        }

                        default:
                        {
                                // Unknown frame type so declare error
                                mac_error ("Unexpected frame type received.",
OPC_NIL, OPC_NIL);
                        }

                } // End switch that splits the flow according the type of the
```

```
received frame

                        break;

                } // End case expectedFrameType = UanE_Cts

                // The node expects a data frame
                case (UanE_Data):
                {
                        // Depending on the frame type received
                        switch (rcvdDummyHeader->frameType)
                        {

                                // If received a RTS, or a Cts, do nothing and wait for
timeout
                                case (UanE_Rts):
                                case (UanE_Cts):
                                {
                                        break;
                                }

                                // If the received type frame is a data frame
                                case (UanE_Data):
                                {
                                        updateDataTrafficStats(rcvd_pk_size, rx_start_time);

                                        // Process frame only if it destined for this node
and from the correct node
                                        if (rcvdDummyHeader->receiverAddress == myAddress &&
rcvdDummyHeader->senderAddress == remoteAddress)
                                        {

                                                // Update NAV and set the updated flag
                                                navTime = rcvdDummyHeader->senderTime +
rcvdDummyHeader->reservedDuration;
                                                flags->nav_updated;

                                                // Retrieve the fragment
                                                op_pk_fd_get (rcvd_frame_ptr, 1,
&seg_pkptr);

                                                // Call the processing routine
                                                data_process (seg_pkptr, rcvdDummyHeader);
                                        }

                                        // If not destined to this node do nothing and wait
for timeout
                                        else
                                        {}

                                        break;

                                } // End the case where the expected and received type is a
data frame

                                // If receive an ACK, do nothing and wait for timeout
                                case (UanE_Ack):
                                {
                                        break;
                                }

                        } // End switch that splits the flow according the type of the
received frame

                        break;

                } // End case expectedFrameType = UanE_Data

                // The expected frame in an Ack
                case (UanE_Ack):
                {
```

```
                        // Split the flow according the receiving frame
                        switch (rcvdDummyHeader->frameType)
                        {

                                // In all other cases do nothing and wait timeout
                                case (UanE_Rts):
                                case (UanE_Cts):
                                case (UanE_Data):
                                {
                                        break;
                                }

                                // The received frame is an Ack
                                case (UanE_Ack):
                                {
                                        updateControlTrafficStats (rcvd_pk_size,
rx_start_time);

                                        printf("\n#%s# - Receiver address = %i; Sender
address = %i", myName, rcvdDummyHeader->receiverAddress,rcvdDummyHeader->senderAddress);
                                        printf("\n#%s# - My address = %i; Remote address =
%i", myName, myAddress,remoteAddress);
                                        printf("\nAck fragments: %x", rcvdDummyHeader-
>ackFragments);

                                        // Check if the ACK is for this node
                                        if (rcvdDummyHeader->receiverAddress == myAddress &&
                                                rcvdDummyHeader->senderAddress ==
destinationAddress)
                                        {

                                                printf("\n#%s# - dataPacketID = %i", myName,
rcvdDummyHeader->dataPacketID);
                                                printf("\n#%s# - packetInService = %i",
myName, packetInService);

                                                // Check if the ACK refers to the data
packet sent
                                                if (rcvdDummyHeader->dataPacketID ==
packetInService)
                                                {
                                                        printf("\nBefore receiving ack. Ack
Fragments: ");
                                                        for (idx = 0; idx < 16; idx++)
                                                        {
                                                                printf("%x ",
rcvdDummyHeader->ackFragments[idx]);
                                                        }

                                                        printf("\n#%s# - Before for num
fragments = %i", myName, rcvdDummyHeader->dataPacketID);
                                                        printf("\nAck fragments: ");

                                                        /* Determine which of the fragments
were acked and destroy the
                                                           fragments successfully acked */
                                                        for (idx = 0; idx < op_prg_list_size
(fragmentationList); idx++)
                                                        {
                                                                sentFragment =
(UanT_Mac_Fragmentation_List_Element*)
                                                        op_prg_list_access
(fragmentationList, idx);

                                                                        if (rcvdDummyHeader-
>ackFragments[idx] == OPC_TRUE)
                                                                        {
                                                        printf("\n#%s# - Inside deleting
fragment: %i", myName, sentFragment->fragmentNumber);
                                                                                sentFragment-
>transmitted = OPC_TRUE;
```

```
                                                                }
                                                                else
                                                                {
                                                                        sentFragment-
>transmitted = OPC_FALSE;
                                                                }

                                                        printf("\n#%s - Flag
transmitted of fragment %i of packet ID %i set to %s", myName, sentFragment-
>fragmentNumber, packetInService, ((sentFragment->transmitted == OPC_TRUE) ? "TRUE" :
"FALSE"));
                                                }


                                                // Check whether all the data packet
was acked
                                                allFragmentsTransmitted = 0;
                                                for (idx = 0; idx < op_prg_list_size
(fragmentationList); idx++)
                                                {

                                                        if (sentFragment->transmitted
== OPC_TRUE)

        allFragmentsTransmitted++;
                                                }

                                                if (op_prg_list_size
(fragmentationList) == allFragmentsTransmitted)
                                                {
                                                        for (idx = op_prg_list_size
(fragmentationList); idx > 0; idx--)
                                                        {
                                                                sentFragment =
(UanT_Mac_Fragmentation_List_Element*)

        op_prg_list_remove(fragmentationList, idx - 1);
                                                                op_prg_mem_free
(sentFragment);
                                                        }


                                                        op_stat_write (retransHandle,
(double) (retryCount * 1.0));

        retransmissionAttemptsByNode[myAddress] += retryCount;

                                                        // Reset the retry counter as
data packet is successfully acked
                                                        retryCount = 0;

                                                        // The fragmentation buffer
is empty
                                                        flags->fragments_to_send =
OPC_FALSE;

                                                        // Printing out information
to ODB
                                                        printf ("\n#%s# - Ack
received for data packet " OPC_PACKET_ID_FMT,
                                                                                myName,
packetInService);
                                                }

                                                // If partial acking
                                                else
                                                {
                                                        /* Increment the
retransmission counter and check whether further
                                                           retries are possible or
the packet, or its remains, needs to be discarded */
```

```
                                                        if (networkMode ==
UanE_Contention_Based)
                                                        {
                                                                retryCount++;
                                                                frame_discard ();
                                                        }
                                                        else
                                                                retryCount = 0;

                                                        // Printing out information
to ODB
                                                        printf ("\n#%s# - Incompleted
Ack received for data packet "
        OPC_PACKET_ID_FMT, myName, packetInService);
                                                }
                                        }

                                        /* Received an ACK but for a different data
packet. This means that we
                                           may consider that the data packet was not
received and therefore we
                                           need to start the all process again */
                                        else
                                        {
                                                /* Increment the retransmission
counter and check whether further
                                                   retries are possible or the
packet, or its remains, needs to be discarded */
                                                retryCount++;
                                                frame_discard ();

                                                // Printing out information to ODB
                                                printf("\n#%s# - Expecting an ACK,
and received an ACK for a "
                                                        "different data
packet", myName);
                                        }

                                        /* No matter all fragments were acked, the
node needs to recontent for the
                                           channel */
                                        flags->rts_sent = OPC_FALSE;

                                        /* After a period where the node had the
channel it needs to perform
                                           backoff */
                                        flags->backoff_required = OPC_TRUE;


                                        // Schedule a resume timeout event
                                        op_intrpt_schedule_self(op_sim_time(),
UanE_Resume_Timeout);

                                        // The frame to send
                                        frameTypeToSend = UanE_None;

                                        // The expected frame
                                        expectedFrameType = UanE_None;

                                }

                                /* If the ACK is not for this node and from the
expected remote station wait
                                   for the timeout */
                                else
                                {
                                        // Printing out information to ODB
                                        printf ("\n#%s# - The expected ACK is not
received.", myName);
                                }
```

```
                                        break;

                                } // End case where the expected and received type frame is
an Ack

                                default:
                                {
                                        // Unknown frame type so declare error
                                        mac_error ("Unexpected frame type received.",
OPC_NIL, OPC_NIL);
                                }

                                // No matter whether the Ack is destined to this node, I
need to update the navTime
                                navTime = rcvdDummyHeader->senderTime + rcvdDummyHeader-
>reservedDuration;

                        } // End switch that splits the flow according the type of the
received frame

                        break;

                } // End case expectedFrameType = UanE_Ack

                default:
                {
                        // Unknown frame type so declare error
                        mac_error ("Unexpecting unknown frame type.", OPC_NIL, OPC_NIL);
                }

        } // end switch expectedFrameType


        /* Report the amount of time the channel will be busy if the NAV is updated with
the
           received packet */
        if (navTime - previous_nav > PRECISION_RECOVERY)
        {
                op_stat_write (channelReservHandle, (navTime - previous_nav));
                op_stat_write (channelReservHandle, 0.0);
        }

        // Destroying the received frame once relevant information is taken out of it
        op_pk_destroy (rcvd_frame_ptr);


        FOUT;
}

/*
##############################################################################################
#########################
##############################################################################################
#########################
No further retries for the data frame for which the retry limit has reached. As a result
these frames are
discarded
*/
static void frame_discard ()
{
        int fragmentsListSize;
        UanT_Mac_Fragmentation_List_Element* packetFragment;
        int idx;

        FIN (frame_discard ());

        // If retry limit has reached then drop the frame
        if (retryCount == retryLimit)
        {
                // Update retransmission count statistic
```

```
                op_stat_write (retransHandle, (double) (retryCount * 1.0));
                retransmissionAttemptsByNode[myAddress] += retryCount;

                // Update the local and global dropped packet statistics
                op_stat_write (dropPacketHandle, 1.0);
//          op_stat_write (dropPacketHandle, 0.0);
                op_stat_write (dropPacketHandleInBits, (double) currentPacketSize);
            op_stat_write (dropPacketHandleInBits, 0.0);
                op_stat_write (globalDroppedDataHandle, (double) currentPacketSize);
                op_stat_write (globalDroppedDataHandle, 0.0);

                // Update the dropped packet statistics
                packetsDroppedByNode[myAddress]++;
                packetsInQueueByNode[myAddress]--;

                // Reset the retry count for the next packet
                retryCount = 0;

                /* Get the segmentation buffer size to check if there are more fragments
left to
                   be transmitted */
                fragmentsListSize =  op_prg_list_size (fragmentationList);

                for (idx = fragmentsListSize; idx > 0; idx--)
                {
                        packetFragment = (UanT_Mac_Fragmentation_List_Element*)
                                        op_prg_list_remove (fragmentationList, idx -
1);
                        op_prg_mem_free (packetFragment);

                        flags->fragments_to_send = OPC_FALSE;
                }

                // Reset the "frame to respond" variable unless we have a CTS or ACK to
send
//              if (frameTypeToSend == UanE_Data)
                        frameTypeToSend = UanE_None;

                /* If there is not any other data packet sent from higher layer and waiting
in the buffer for
                   transmission, reset the related flag */
                if (op_prg_list_size (highLayerListPtr) == 0)
                        flags->packet_to_send = OPC_FALSE;

                /* Although we could not transmit this data packet and eventually dropped
it, still set the contention
                   window flag and back-off for a contention window period. This is
necessary for the fairness of the
                   algorithm. This prevents us going to IDLE state (if higher layer data
queue is empty) and then may
                   attempt to transmit a packet without waiting for a full backoff period
as a result of suddenly
                   receiving a packet from higher layer */
                flags->backoff_required = OPC_TRUE;
        }

        FOUT;
}


/*
##############################################################################################
#########################
##############################################################################################
#########################
This routine handles defragmentation process and also sends data to the higher layer if
all the fragments have
been received by the station
*/
static void data_process (Packet * seg_pkptr, UanT_Frame_Fields* rcvdDummyHeader)
{
```

```
	int
	current_index;
	int
	list_index;
	int
	list_size;
	int idx;
	OpT_Packet_Size
	pkt_size;
	UwnT_Mac_Defragmentation_Buffer_Entry*     defrag_ptr = OPC_NIL;
	Objid sourceModuleObjID;
	Objid sourceNodeObjID;
	Objid sourceMacLayerObjID;
	char sourceNodeName[25];
	int sourceMacAddress;

	FIN (data_process (seg_pkptr, rcvdDummyHeader));

	if (networkMode == UanE_Aloha_Alike)
		remoteAddress = rcvdDummyHeader->senderAddress;

       printf("\n#%s# Beginning data process. Packet ID " OPC_PACKET_ID_FMT, myName,
rcvdDummyHeader->dataPacketID);

           /* The original packet is being transmitted in multiple fragments. Insert
fragments into the reassembly
              buffer. There are two possible cases:
              1. The remote station has just started sending the fragments and it
doesn't exist in the list
              2. The remote station does exist in the list and the and the new
fragment is a series of fragments for
                  the data packet */
            printf("\n#%s# Inside data process", myName);
            // Get the size of the defragmentation list
            list_size = op_prg_list_size (defragmentationListPtr);

            /* Initialize the current node index which will indicate whether the entry
for the station exists in the
              list */
            current_index = -1;


            printf("\n#%s# - My remote address is %i", myName, remoteAddress);
            /* Searching through the list to find if the remote station address exists
i.e. the source station has
              received fragments for this data packet before. Also, removing entries
from the defragmentation buffer
              which has reached its maximum receive lifetime */
            for (list_index = 0; list_index < list_size; )
            {
                    // Accessing node of the list for search purposes
                    defrag_ptr = (UwnT_Mac_Defragmentation_Buffer_Entry *)
op_prg_list_access (defragmentationListPtr,

                                                                        list_index);

                    // If the station entry already exists in the list then store its
index for future use

                    if ( remoteAddress == defrag_ptr->tx_station_address)
                    {
                            if (defrag_ptr->dataPacketID == rcvdDummyHeader-
>dataPacketID)
                            {
                                    current_index = list_index;
                                    currentIndexDefragmentationBuffer = current_index;
                                    printf("\n#%s# - The remote address exists in the
buffer. The index is %i", myName, currentIndexDefragmentationBuffer);
                            }
                            else
                            {
```

```
                                        defrag_ptr =(UwnT_Mac_Defragmentation_Buffer_Entry
*) op_prg_list_remove (defragmentationListPtr,

list_index);
                                        op_sar_buf_destroy (defrag_ptr-
>reassembly_buffer_ptr);
                                        op_prg_mem_free (defrag_ptr);
                                        list_size--;

                                }

                                // Exit the loop since we have found the entry we were
looking for
                                list_index = list_size;
                        }

                        // Otherwise move to the next element in the list
                        else
                                list_index++;
                }

                // If remote station entry doesn't exist then create new node
                if (current_index == -1)

                {
                        /* If the entry of the station does not exist in the defrag list
and the fragment received is not the
                           first fragment of the packet then it implies that the maximum
receive lifetime of the packet has
                           expired. In this case the received packet will be destroyed and
the acknowledgement is sent to the
                           receiver as specified by the protocol */

                        // Creating struct type for defragmentation structure
                        defrag_ptr = (UwnT_Mac_Defragmentation_Buffer_Entry *)
                                        op_prg_mem_alloc (sizeof
(UwnT_Mac_Defragmentation_Buffer_Entry));

                        // Generate error and abort simulation if no more memory left to
allocate for duplicate buffer
                        if (defrag_ptr == OPC_NIL)
                                mac_error ("Cannot allocate memory for defragmentation
buffer entry", OPC_NIL, OPC_NIL);

                        // Source station address is store in the list for future reference
                        defrag_ptr->tx_station_address = rcvdDummyHeader->senderAddress;

                        // Initialize the number of fragments received
                        defrag_ptr->receivedNumberOfFragments = 0;

                        // Initialize ackFragments field
                        printf("\n#%s# - TESTE - Ack fragments = ");
                        for (idx = 0; idx < 16; idx++ )
                        {
                                defrag_ptr->ackFragments[idx] = OPC_FALSE;
                                printf("%x ", defrag_ptr->ackFragments[idx]);
                                }

                        // Initialize sent to higher layer flag
                        defrag_ptr->sentHigherLayer = OPC_FALSE;

                        // For new node creating a reassembly buffer
                        defrag_ptr->reassembly_buffer_ptr = op_sar_buf_create
(OPC_SAR_BUF_TYPE_REASSEMBLY,

OPC_SAR_BUF_OPT_DEFAULT);
                        op_prg_list_insert (defragmentationListPtr, defrag_ptr,
OPC_LISTPOS_TAIL);

                        currentIndexDefragmentationBuffer = op_prg_list_size
(defragmentationListPtr) - 1;
```

```
		}

		if (defrag_ptr == OPC_NIL)
			printf("\n#%s# POINTER NULO!?!?!?!", myName);

	printf("\n#%s# Before adding to defrag buffer on frmae header. Packet ID: "
OPC_PACKET_ID_FMT, myName, rcvdDummyHeader->dataPacketID);


		// Check whether the fragment was already received, before adding it to
the reassembly buffer
		if (defrag_ptr->ackFragments[rcvdDummyHeader->fragmentNumber] ==
OPC_FALSE)
		{
			// Record the received time of this fragment
			defrag_ptr->time_rcvd = currentTime;
			printf("\n#%s# Inside adding the fragment to the buffer. Received
fragment: %i",myName, rcvdDummyHeader->fragmentNumber);

			defrag_ptr->dataPacketID = rcvdDummyHeader->dataPacketID;

			defrag_ptr->ackFragments[rcvdDummyHeader->fragmentNumber] =
OPC_TRUE;

			// Insert fragment into the reassembly buffer
			op_sar_rsmbuf_seg_insert (defrag_ptr->reassembly_buffer_ptr,
seg_pkptr);
			defrag_ptr->receivedNumberOfFragments++;
		}

		printf("\n#%s# After adding to defrag buffer " OPC_PACKET_ID_FMT, myName,
defrag_ptr->dataPacketID);

		printf("\n#%s# - Outside reassembly last fragment\n", myName);
		printf("\n#%s# - number of fragments in header = %i\n", myName,
rcvdDummyHeader->numberOfFragments);
		printf("\n#%s# - number of fragments in defrag pointer = %i\n", myName,
defrag_ptr->receivedNumberOfFragments);

	// If this is the last fragment then send the data to higher layer
	if (rcvdDummyHeader->numberOfFragments == defrag_ptr->receivedNumberOfFragments)
	{
		printf("\n#%s# - Inside reassembly last fragment\n", myName);

			frameTypeToSend = UanE_Ack;
			expectedFrameType = UanE_None_Transit;
			op_intrpt_schedule_self (op_sim_time(), UanE_Resume_Timeout);

		// Check whether the packet was already forwarded to the higher layer
		if (defrag_ptr->sentHigherLayer == OPC_FALSE)
		{
			defrag_ptr->sentHigherLayer = OPC_TRUE;
			seg_pkptr = op_sar_rsmbuf_pk_remove (defrag_ptr-
>reassembly_buffer_ptr);

			switch (typeOfNode)
			{
				case UanE_Relay_Node:
				{
					printf("\n#%s# - I am a relay node", myName);
					pkt_size = op_pk_total_size_get (seg_pkptr);
					packetsInQueueByNode[rcvdDummyHeader-
>senderAddress]--;

					// If buffer is too full to accept the packet, drop
it and report to statistics
					if (highLayerListTotalSize + pkt_size >
highLayerListMaxSize)
						high_layer_packet_drop (seg_pkptr);
```

```
                                        // otherwise update the buffer size and enqueue the
packet
                                        else
                                        {
                                                highLayerListTotalSize += pkt_size;

                                                /* Update the local/global throughput and
end-to-end delay statistics based on the packet that will be
                                                   forwarded to the higher layer */


                                                printf("\n#%s# Relay node before stats.
Stamped packet in stats %f", myName, op_pk_stamp_time_get(seg_pkptr));
                                                printf("\n and the current time = %f",
currentTime);

                                                printf("\n and the delay = %f", currentTime
- op_pk_stamp_time_get (seg_pkptr));

                                                sourceModuleObjID = op_pk_creation_mod_get
(seg_pkptr);

                                                sourceNodeObjID = op_topo_parent
(sourceModuleObjID);
                                                sourceMacLayerObjID = op_id_from_name
(sourceNodeObjID, OPC_OBJTYPE_QUEUE, "MAC Layer");

                                                op_ima_obj_attr_get (sourceMacLayerObjID,
"MAC Address", &sourceMacAddress);


                                                accepted_frame_stats_update (seg_pkptr,
sourceMacAddress, rcvdDummyHeader->typeOfTraffic);

                                                printf("\n#%s# - Forwarding the packet to
the high layer queue", myName);

                                                high_layer_packet_enqueue (seg_pkptr,
rcvdDummyHeader->typeOfTraffic);

                                                // Update packets generated statistics
                                                packetsInQueueByNode[myAddress]++;


                                        }

                                        break;
                                }

                                case UanE_Sensor_Node:
                                {
                                        printf("\n#%s# - I am a sensor node", myName);
                                        mac_error ("Sensor nodes do not receive data frames.
Terminating simulation.", OPC_NIL, OPC_NIL);
                                        break;
                                }

                                case UanE_Gateway_Node:
                                {
                                        printf("\n#%s# - I am a gateway node", myName);

                                        sourceModuleObjID = op_pk_creation_mod_get
(seg_pkptr);

                                        sourceNodeObjID = op_topo_parent
(sourceModuleObjID);
                                        sourceMacLayerObjID = op_id_from_name
(sourceNodeObjID, OPC_OBJTYPE_QUEUE, "MAC Layer");

                                        op_ima_obj_attr_get (sourceMacLayerObjID, "MAC
Address", &sourceMacAddress);
```

```
                                        op_ima_obj_attr_get (sourceNodeObjID, "name",
sourceNodeName);

                                        printf ("Packet just received from node [%s]",
sourceNodeName);

                                                printf("\n#%s# Gateway node before stats.
Stamped packet in stats %f", myName, op_pk_stamp_time_get(seg_pkptr));
                                                printf("\n and the current time = %f",
currentTime);

                                                printf("\n and the delay = %f", currentTime
- op_pk_stamp_time_get (seg_pkptr));

                                        packetsInQueueByNode[rcvdDummyHeader-
>senderAddress]--;
                                        lastReceivedPacket[sourceMacAddress] =
op_sim_time();

                                        /* Update the local/global throughput and end-to-end
delay statistics based on the packet that will be
                                           forwarded to the higher layer */
                                        accepted_frame_stats_update (seg_pkptr,
sourceMacAddress, rcvdDummyHeader->typeOfTraffic);

                                        // Sending data to higher layer through mac
interface
                                        op_pk_send (seg_pkptr,
OUTPUT_STREAM_TO_UPPER_LAYER);

                                        break;

                                }

                                default:
                                {
                                        mac_error ("Enable to determine type of node.",
OPC_NIL, OPC_NIL);
                                }
                        printf ("All fragments of Data packet " OPC_PACKET_ID_FMT " is
received and sent to the higher layer", defrag_ptr->dataPacketID);

                        }
                }

        }

        // Otherwise expect more data
        else if (networkMode == UanE_Aloha_Alike)
        {
                frameTypeToSend = UanE_Ack;
                expectedFrameType = UanE_None_Transit;
        }
        else
                expectedFrameType = UanE_Data;


        FOUT;
}

/*

                                */
/*
##############################################################################################
##########################
##############################################################################################
##########################
This function is called just before a frame received from physical layer being forwarded
to the higher layer to
```

```
update end-to-end delay and throughput statistics
*/
static void accepted_frame_stats_update (Packet* seg_pkptr, int address, int traffic)
{
	double  ete_delay, pk_size;

	FIN (accepted_frame_stats_update (seg_pkptr, address, traffic));

	// Total number of bits sent to higher layer is equivalent to a     throughput
	pk_size = (double) op_pk_total_size_get (seg_pkptr);
	printf("\nStatistics - Packet send higher layer size = %f\n", pk_size);
	op_stat_write (throughputHandle, pk_size);
	op_stat_write (throughputHandle, 0.0);

	// Also update the global WLAN throughput statistic
	op_stat_write (globalThroughputHandle, pk_size);
	op_stat_write (globalThroughputHandle, 0.0);

	// Compute the end-to-end delay for the frame and record it
	printf("\n#%s# Stamped packet in stats %f\n", myName,
op_pk_stamp_time_get(seg_pkptr));
	printf("\n and the current time = %f", currentTime);

	ete_delay = currentTime - op_pk_stamp_time_get (seg_pkptr);
	printf("\n and the delay = %f\n", ete_delay);

	if (traffic == UanE_Background_Traffic)
		op_stat_write (eteDelayBackgroundTraffic, ete_delay);

	if (traffic == UanE_Non_Periodic_Traffic)
		op_stat_write (eteDelayNonPeriodicTraffic, ete_delay);

	if (typeOfNode == UanE_Gateway_Node)
	{
		eteDelayAllTrafficByNode[address] += ete_delay;
		allPacketsReceivedByNode[address]++;
		allTrafficReceivedByNode[address] += pk_size;

		if (traffic == UanE_Background_Traffic)
		{
			constTrafficReceivedByNode[address] += pk_size;
			eteDelayConstTrafficByNode[address] += ete_delay;
			constPacketsReceivedByNode[address]++;
		}
		else if (traffic == UanE_Non_Periodic_Traffic)
		{
			eventTrafficReceivedByNode[address] += pk_size;
			eteDelayEventTrafficByNode[address] += ete_delay;
			eventPacketsReceivedByNode[address]++;
		}
	}

	op_stat_write (eteDelayHandle[address], ete_delay);
	op_stat_write (globalETEdelayHandle, ete_delay);

	FOUT;
}


/*##############################################################################################
##
##############################################################################################
##
Main procedure to invoke function for preparing and transmitting the appropriate frames
*/

static void frame_transmit ()
{
	FIN (frame_transmit());

	printf("\n#%s# frame_transmit: type of frame to transmit = %i\n", myName,
```

```
frameTypeToSend);

	// If we are initiating a transmission, prepare the fragments
	if (frameTypeToSend == UanE_Rts)
	{
		// If we are sending fragments of this packet for the first time
		if (flags->packet_to_send == OPC_TRUE && flags->fragments_to_send ==
OPC_FALSE)
		{
			build_packet_fragments_list();
		}

		printf("Packet list size = %i; fragments list size = %i", op_prg_list_size
(highLayerListPtr), op_prg_list_size(fragmentationList));
		printf("Flags Packet = %x; Fragments=%i", flags->packet_to_send, flags-
>fragments_to_send);

		if (networkMode == UanE_Aloha_Alike)
			frameTypeToSend = UanE_Data;
	}


	switch (frameTypeToSend)
	{

		case UanE_Rts:
		{
			/* We may be transmitting the packet for the first time or
retransmitting
			   the fragments of a packet previously fragmented. Then depending
on the
			   size of the fragments to send determine if we need to send an
RTS */

			printf("\n#%s# - Inside frame transmit - the current packet size =
%i", myName, currentPacketSize);


			// Send rts if RTS is enabled and packet size is more than RTS
threshold
				// Prepare RTS frame for transmission
				printf("\n#%s# - In frame_transmit, before sending Rts",
myName);
				prepare_Rts_to_send ();

			break;
		}

		case UanE_Cts:
		{
			printf("\n#%s# - In frame_transmit, before sending Cts", myName);
			prepare_Cts_to_send();
			break;
		}

		case UanE_Data:
		{
			printf("\n#%s# - In frame_transmit, before sending data", myName);

			prepare_data_frame_to_send();
			break;
		}

		case UanE_Ack:
		{
			printf("\n#%s# - In frame_transmit, before sending Ack", myName);
			prepare_Ack_to_send();
			break;
		}

		case UanE_None:
```

```
		default:
		{
			mac_error ("Transmission request for unexpected frame type.",
OPC_NIL, OPC_NIL);
		}

	}

	FOUT;
}

/*
##############################################################################################
##
##############################################################################################
##
*/

static double calculate_NAV_toSend()
{
	double nav;
	double distance;
	int nrBitsInBuffer;
	UanT_Mac_Fragmentation_List_Element* packetFragment;
	int numFragments;
	int idx;

	FIN(calculate_NAV_toSend());

	numFragments = op_prg_list_size (fragmentationList);
	nrBitsInBuffer = 0;

	for (idx = 0; idx < numFragments; idx++)
	{
		packetFragment = (UanT_Mac_Fragmentation_List_Element*)
				op_prg_list_access (fragmentationList, idx);

		if (packetFragment->transmitted == OPC_FALSE)
			nrBitsInBuffer += (int) packetFragment->fragmentSize;
	}

	switch (frameTypeToSend)
	{
		case UanE_Rts:
		{

			nav = TXTIME(sizeCTS) + TXTIME(sizeRTS) + TXTIME(sizeACK) +
				TXTIME((numFragments * sizeDataFrameHeader) +
nrBitsInBuffer) +
				(4.0 * (range / propagationSpeed)) +
(((double)numFragments + 2.0) * sifsDuration);
			break;
		}

		case UanE_Data:
		{
			if (networkMode == UanE_Contention_Based)
			{
				distance = (propagationSpeed / 2.0) *
					(timeCtsReceived - timeRtsSend - TXTIME(sizeRTS) -
TXTIME(sizeCTS) - sifsDuration);
			}
			else
				distance = range;

			printf("\n#%s# - Calculated distance = %f", myName, distance);

			nav = TXTIME((numFragments * sizeDataFrameHeader) + nrBitsInBuffer)
+
				((distance + range) / propagationSpeed) +
				TXTIME(sizeACK) + (numFragments * sifsDuration);
```

```
                        break;
                }

                case UanE_Ack:
                {
                        nav = TXTIME(sizeACK) + (range / propagationSpeed);
                        break;
                }
        }

        FRET(nav);
}



/*
##############################################################################################
##
##############################################################################################
##
Function to retrieve the name of the destination node from the destination address
defined
during the network configuration. The name will be used later to "send" the frames
*/
static void findDestinationNodeName()
{
        Objid tempNode;
        Objid tempProc;
        int i;
        int nrNodes;
        int tempDest;

        FIN(findDestinationNodeName());
        printf("\n#%s# - Inside findDestinationNodeName()", myName);

        // Retrieving the total number of nodes
        nrNodes = op_topo_child_count (myNetworkObjectID, OPC_OBJMTYPE_NODE);

        printf("\n#%s# - Inside findDestinationNodeName() - nrNodes in the network = %i",
myName, nrNodes);

        /* Traversing the nodes in the network to find the one with the destination
address and retrieving the name of
           that node */
        for (i=0; i < nrNodes; i++)
        {
                tempNode = op_topo_child (myNetworkObjectID, OPC_OBJMTYPE_NODE, i);
                tempProc = op_id_from_name (tempNode, OPC_OBJTYPE_QUEUE, MAC_LAYER);
                op_ima_obj_attr_get (tempProc, "MAC Address", &tempDest);
                if (tempDest == destinationAddress)
                {
                        op_ima_obj_attr_get (tempNode, "name", destinationNodeName);
                        printf("Destination node name %s\n", destinationNodeName);
                        FOUT;
                }
        }

        FOUT;
}

/*
##############################################################################################
##
##############################################################################################
##
Function that determines if the received frame should be considered an error frame
according to the defined error rate
*/
static OpT_Boolean errorFrame()
{
        double random;
```

```
	FIN (determineIfErrorFrame());

	random = op_dist_outcome (errorRateDistribution);

	printf("\n#%s# - Error frame", myName);

	if (random <= (errorRate / DOUBLE_ONE_HUNDRED))
		{FRET(OPC_TRUE);}
	else
		FRET(OPC_FALSE);
}


/*
                                                                                */
/*
##############################################################################################
##
##############################################################################################
##
Prepare Acks to transmit by setting appropriate fields in the dummy header
*/
static void prepare_Ack_to_send ()
{
//	double			mac_delay;
	double			total_pk_size;
	double				tx_end_time;
	UanT_Frame_Fields* dummyFrameHeader;
	Packet*			transmit_frame_ptr;
	UwnT_Mac_Defragmentation_Buffer_Entry*	defrag_ptr = OPC_NIL;
	int idx;

	FIN (prepare_Ack_to_send ());

	// Accessing node of the list for search purposes
	defrag_ptr = (UwnT_Mac_Defragmentation_Buffer_Entry *) op_prg_list_access
(defragmentationListPtr, currentIndexDefragmentationBuffer);

	// And reset the index
	currentIndexDefragmentationBuffer = -1;

	// Creating a frame with the ACK size
	transmit_frame_ptr = op_pk_create (sizeACK);

	printf("\n#%s# Creating an Ack for data packet " OPC_PACKET_ID_FMT, myName,
defrag_ptr->dataPacketID);


	dummyFrameHeader = ((UanT_Frame_Fields*) op_prg_mem_alloc
(sizeof(UanT_Frame_Fields)));

	// Setting the fields in the dummy frame header
	dummyFrameHeader->frameType = UanE_Ack;
	dummyFrameHeader->senderAddress = myAddress;
	dummyFrameHeader->receiverAddress = remoteAddress;
	dummyFrameHeader->dataRate = outboundChannelDataRate;
	dummyFrameHeader->dataPacketID = defrag_ptr->dataPacketID;

	for (idx = 0; idx < 16; idx++)
	{
		dummyFrameHeader->ackFragments[idx] = defrag_ptr->ackFragments[idx];
		}


	//
	navTime = calculate_NAV_toSend() + currentTime;
	dummyFrameHeader->senderTime = currentTime;
	dummyFrameHeader->reservedDuration = navTime - currentTime;

	// Since no frame is expected, the expected frame type field to nil
```

```
	expectedFrameType = UanE_None;

	/* Once Ack is transmitted in response to Data frame then set the frame response
	   indicator to none frame as the response is already generated     */
	frameTypeToSend = UanE_None;

	// Set the last frame state variable
	lastFrameTxType = UanE_Ack;


	// Adding the dummy header to the frame (this action does not affect the frame
size)
	op_pk_fd_set (transmit_frame_ptr, 0, OPC_FIELD_TYPE_STRUCT, dummyFrameHeader, 0,
op_prg_mem_copy_create, op_prg_mem_free, sizeof (UanT_Frame_Fields));

	// Update the control traffic sent statistics
	total_pk_size = (double) op_pk_total_size_get (transmit_frame_ptr);
	op_stat_write (ctrlTrafficSentHandleInBits, total_pk_size);
	op_stat_write (ctrlTrafficSentHandle, 1.0);

	// Write a value of 0 for the end of transmission
	tx_end_time = currentTime + TXTIME(sizeACK);
	op_stat_write_t (ctrlTrafficSentHandleInBits, 0.0, tx_end_time);
	op_stat_write_t (ctrlTrafficSentHandle,        0.0, tx_end_time);

	// Send packet to the transmitter
	op_pk_send (transmit_frame_ptr, OUTPUT_STREAM_TO_PHYSICAL_LAYER);
	flags->transmitter_busy = OPC_TRUE;

	printf("\n#%s# - Sending Ack - TRANSMITTER ON", myName);

	// If this is a relay node, after sending the ACK we should set the type of frame
to send
	if (typeOfNode == UanE_Relay_Node && NEED_TO_TRANSMIT)
		frameTypeToSend = UanE_Rts_Ime;

	FOUT;

} // end prepare_Ack_send()


/*
                                                                                */
/*
###############################################################################################
###############################################################################################
##
Prepare a Cts frame to send
*/
static void prepare_Cts_to_send ()
{
	double                  total_pk_size;
	double                  tx_end_time;
	UanT_Frame_Fields* dummyFrameHeader;
	Packet*                 transmit_frame_ptr;

	FIN (prepare_Cts_to_send ());

	dummyFrameHeader = ((UanT_Frame_Fields*) op_prg_mem_alloc
(sizeof(UanT_Frame_Fields)));

	// Creating a frame with the CTS size
	transmit_frame_ptr = op_pk_create(sizeCTS);

	printf("\n#%s# Creating a CTS that will be transmitted with ID: %i\n", myName,
(int)op_pk_id(transmit_frame_ptr));


	// Setting the fields in the dummy frame header
	dummyFrameHeader->frameType = UanE_Cts;
	dummyFrameHeader->senderAddress = myAddress;
```

```
        dummyFrameHeader->receiverAddress = remoteAddress;
        dummyFrameHeader->dataRate = outboundChannelDataRate;

        /* Station is reserving channel bandwidth by using RTS frame, so in RTS the
station
           will broadcast the duration it needs to send the data and receive ACK for it.
           Because we are using absolute time, we just need to broadcast the same navTime
*/
        dummyFrameHeader->senderTime = currentTime;
        dummyFrameHeader->reservedDuration = navTime - currentTime;

        // Adding the dummy header to the frame (this action does not affect the frame
size)
        op_pk_fd_set (transmit_frame_ptr, 0, OPC_FIELD_TYPE_STRUCT |
OPC_FIELD_SIZE_IS_INT64, dummyFrameHeader, OPC_FIELD_SIZE_UNCHANGED,
op_prg_mem_copy_create, op_prg_mem_free, sizeof (UanT_Frame_Fields));

        /* Setting the variable which keeps track of the last transmitted frame that needs
           response */
        lastFrameTxType = UanE_Cts;

        /* Once CTS is transmitted in response to RTS then set the frame response
indicator to
           none frame as the response is already generated */
        frameTypeToSend = UanE_None;

        // The expected frame once CTS is transmitted
        expectedFrameType = UanE_Data;

        // Update the control traffic sent statistics
        total_pk_size = (double) op_pk_total_size_get (transmit_frame_ptr);
        op_stat_write (ctrlTrafficSentHandleInBits, total_pk_size);
        op_stat_write (ctrlTrafficSentHandle,               1.0);

        /* Write a value of 0 for the end of transmission.
        */
        tx_end_time = currentTime + TXTIME(sizeCTS);
        op_stat_write_t (ctrlTrafficSentHandleInBits, 0.0, tx_end_time);
        op_stat_write_t (ctrlTrafficSentHandle,               0.0, tx_end_time);

        // Send packet to the transmitter
        op_pk_send (transmit_frame_ptr, OUTPUT_STREAM_TO_PHYSICAL_LAYER);
        flags->transmitter_busy = OPC_TRUE;
        printf("\n#%s# - Sending Cts - TRANSMITTER ON", myName);

        FOUT;

} // end prepare_Cts_to_send()


/*
                                                                                */
/*
###############################################################################################
###############################################################################################
##
Prepare a Rts frame to send
*/
static void prepare_Rts_to_send ()
{
        double                                          total_pk_size;
        double                                          tx_end_time;
        UanT_Frame_Fields* dummyFrameHeader;
        Packet*                                         transmit_frame_ptr;

        FIN (prepare_Rts_to_send ());

        dummyFrameHeader = ((UanT_Frame_Fields*) op_prg_mem_alloc
(sizeof(UanT_Frame_Fields)));

        // Creating a frame with the RTS size
```

```
        transmit_frame_ptr = op_pk_create(sizeRTS);

        printf("\n#%s# Creating a RTS that will be transmitted with ID: %i\n", myName,
(int)op_pk_id(transmit_frame_ptr));


        // Setting the fields in the dummy frame header
        dummyFrameHeader->frameType = UanE_Rts;
        dummyFrameHeader->senderAddress = myAddress;
        dummyFrameHeader->receiverAddress = destinationAddress;
        dummyFrameHeader->dataRate = outboundChannelDataRate;

        // Station update of its own nav_duration
        navTime = calculate_NAV_toSend() + currentTime;

        /* Station is reserving channel bandwidth by using RTS frame. It will broadcast
the
           duration it needs to send the data and receive ACK for it */
        dummyFrameHeader->senderTime = currentTime;
        dummyFrameHeader->reservedDuration = navTime - currentTime;

        // Adding the dummy header to the frame (this action does not affect the frame
size)
        op_pk_fd_set (transmit_frame_ptr, 0, OPC_FIELD_TYPE_STRUCT |
OPC_FIELD_SIZE_IS_INT64, dummyFrameHeader, OPC_FIELD_SIZE_UNCHANGED,
op_prg_mem_copy_create, op_prg_mem_free, sizeof (UanT_Frame_Fields));
/*
                                                                                */

        /* Setting the variable which keeps track of the last transmitted frame that needs
           response */
        lastFrameTxType = UanE_Rts;


        // CTS is expected in response to RTS
        expectedFrameType = UanE_Cts;

        frameTypeToSend = UanE_None;

        // Set the time when the Rts is send
        timeRtsSend = currentTime;

        // Update the control traffic sent statistics
        total_pk_size = (double) op_pk_total_size_get (transmit_frame_ptr);
        op_stat_write (ctrlTrafficSentHandleInBits, total_pk_size);
        op_stat_write (ctrlTrafficSentHandle,               1.0);

        // Write a value of 0 for the end of transmission
        tx_end_time = currentTime + TXTIME(sizeRTS);
        op_stat_write_t (ctrlTrafficSentHandleInBits, 0.0, tx_end_time);
        op_stat_write_t (ctrlTrafficSentHandle,        0.0, tx_end_time);

        // Send packet to the transmitter
        op_pk_send (transmit_frame_ptr, OUTPUT_STREAM_TO_PHYSICAL_LAYER);
        flags->transmitter_busy = OPC_TRUE;
        printf("\n#%s# - Sending Rts - TRANSMITTER ON", myName);

        FOUT;

} // end prepare_Rts_to_send()


/*
                                                                                */

/*
###############################################################################################
##
###############################################################################################
##
Prepare data frame to transmit by setting appropriate fields in the dummy header
```

```
*/
static void prepare_data_frame_to_send ()
{
	Packet*                                                 seg_pkptr;
	double                                                  total_pk_size;
	double                                                  tx_end_time;
	UanT_Frame_Fields* dummyFrameHeader;
	Packet*                                                 transmit_frame_ptr;
	int allFragmentsTransmitted;
	int idx;
	UanT_Mac_Fragmentation_List_Element* packetFragment;

	FIN (prepare_data_frame_to_send ());

	dummyFrameHeader = ((UanT_Frame_Fields*) op_prg_mem_alloc
(sizeof(UanT_Frame_Fields)));

	// Setting the fields in the dummy frame header
	dummyFrameHeader->frameType = UanE_Data;
	dummyFrameHeader->originatorAddress = myAddress;
	dummyFrameHeader->senderAddress  = myAddress;
	dummyFrameHeader->receiverAddress = destinationAddress;
	dummyFrameHeader->dataRate = outboundChannelDataRate;
	dummyFrameHeader->dataPacketID = packetInService;

	navTime = calculate_NAV_toSend() + currentTime;
	dummyFrameHeader->senderTime = currentTime;
	dummyFrameHeader->reservedDuration = navTime - currentTime;


	printf("\n#%s# - Building a frame. Packet ID %i - the dest address on dummy header
= %i", myName, dummyFrameHeader->dataPacketID, dummyFrameHeader->receiverAddress);

	printf("\n#%S# Inside the while in prepare data frame. The frgaments list size =
%i", myName, op_prg_list_size(fragmentationList));

	/* Remove next fragment from the fragmentation buffer for transmission and set the
	appropriate fragment number */
	allFragmentsTransmitted = 0;
	for (idx = 0; idx < op_prg_list_size (fragmentationList); idx++)
	{
		packetFragment = (UanT_Mac_Fragmentation_List_Element*)
				op_prg_list_access (fragmentationList, idx);
			if (packetFragment->fragment == OPC_NIL)
				{
				printf("\n O fragment na lista nao tem nada????");
				}
				else
					{
					printf("\n O fragment na lista esta bom. E a
transmitted flag = %s", packetFragment->transmitted == OPC_TRUE ? "TRUE" : "FALSE");
					}


		if (packetFragment->transmitted == OPC_TRUE)
		{
			allFragmentsTransmitted++;
		}
		else
		{
			printf("\n#%s# - Inside copying fragments", myName);
			seg_pkptr = op_pk_copy (packetFragment->fragment);
			if (seg_pkptr == OPC_NIL)
				{
				printf("\n O packet nao tem nada????");
				}
				else
					{
					printf("\n O packet esta bom");
					}
```

```
                        if (networkMode == UanE_Contention_Based)
                                packetFragment->transmitted = OPC_TRUE;

                        allFragmentsTransmitted++;

                        break;
                }


        }


        /* If this is the last fragment in the fragmentation list to be transmitted, then
set
           the appropriate flags to wait for ackowledgment */
        if ((allFragmentsTransmitted == op_prg_list_size (fragmentationList) &&
networkMode == UanE_Contention_Based) ||
                networkMode == UanE_Aloha_Alike)
        {

                frameTypeToSend = UanE_None;
                expectedFrameType = UanE_Ack;
        }

        printf("\n#%s# Retrieving the segment from the fragmentation buffer with ID:
%i\n", myName, (int)op_pk_id(seg_pkptr));

        // Set fragment number in packet field and type of traffic
        dummyFrameHeader->fragmentNumber = packetFragment->fragmentNumber;
        dummyFrameHeader->typeOfTraffic = packetFragment->typeOfTraffic;
        dummyFrameHeader->numberOfFragments = packetFragment->numberOfFragments;

                // Printing out information to ODB
        printf ("\n#%s# - Data fragment %d for packet " OPC_PACKET_ID_FMT " is "
             "transmitted", myName, packetFragment->fragmentNumber, packetInService);


        // Set the frame and send the packet to the transmitter
        transmit_frame_ptr = op_pk_create (sizeDataFrameHeader);

        printf("\n#%s# Creating the frame that will be transmitted with ID: %i\n", myName,
(int)op_pk_id(transmit_frame_ptr));

        op_pk_fd_set (transmit_frame_ptr, 0, OPC_FIELD_TYPE_STRUCT |
OPC_FIELD_SIZE_IS_INT64, dummyFrameHeader, OPC_FIELD_SIZE_UNCHANGED,
op_prg_mem_copy_create, op_prg_mem_free, sizeof (UanT_Frame_Fields));

        op_pk_fd_set (transmit_frame_ptr, 1, OPC_FIELD_TYPE_PACKET, seg_pkptr, -1);

        printf ("\n#%s# - Send frame with size %i to the lower layer.", myName,
op_pk_total_size_get(transmit_frame_ptr));
        printf ("\n Start transmission = %f", currentTime);

        op_pk_send (transmit_frame_ptr, OUTPUT_STREAM_TO_PHYSICAL_LAYER);
        flags->transmitter_busy = OPC_TRUE;
        lastFrameTxType = UanE_Data;

        printf("\n#%s# - Sending Data - TRANSMITTER ON", myName);

        // Update the data traffic sent statistics
        total_pk_size = (double) op_pk_total_size_get (transmit_frame_ptr);
        op_stat_write (dataTrafficSentHandleInBits, total_pk_size);
        op_stat_write (dataTrafficSentHandle, 1.0);

        // Write a value of 0 for the end of transmission.                          */
        tx_end_time = currentTime + total_pk_size / outboundChannelDataRate;
        op_stat_write_t (dataTrafficSentHandleInBits, 0.0, tx_end_time);
        op_stat_write_t (dataTrafficSentHandle, 0.0, tx_end_time);

        /* If there is nothing in the higher layer data queue and fragmentation buffer
then
```

```
           disable the data frame flag which will indicate to the station to wait for the
           higher layer packet */
        if (op_prg_list_size (highLayerListPtr) == 0 && op_prg_list_size
(fragmentationList) == 0)
                flags->packet_to_send = OPC_FALSE;

        lastTransmittedPacket[myAddress] = op_sim_time();

        FOUT;
}

static void modelInitialization()
{
        Objid framesSizeCompoundObjID;
        Objid framesSizeObjID;
        Objid inboundChannelsCompoundObjID;
        Objid inboundChannelsObjID;
        Objid frameTransmissionCompoundObjID;
        Objid frameTransmissionObjID;
        Objid contentionCompoundObjID;
        Objid contentionObjID;
        int idx;

        FIN (modelInitialization());

        // object id of the surrounding processor.
        myObjectID = op_id_self ();

        // Obtain the node's object identifier
        myNodeObjectID = op_topo_parent (myObjectID);

        // Obtain the network's object identifier
        myNetworkObjectID = op_topo_parent (myNodeObjectID);

        // Obtain the handle to the compound attributes
        op_ima_obj_attr_get (myObjectID, "Frames Size", &framesSizeCompoundObjID);
        framesSizeObjID = op_topo_child (framesSizeCompoundObjID, OPC_OBJTYPE_GENERIC, 0);

        op_ima_obj_attr_get (myObjectID, "Inbound Channels",
&inboundChannelsCompoundObjID);
        inboundChannelsObjID = op_topo_child (inboundChannelsCompoundObjID,
OPC_OBJTYPE_GENERIC, 0);

        op_ima_obj_attr_get (myObjectID, "Frame Transmission",
&frameTransmissionCompoundObjID);
        frameTransmissionObjID = op_topo_child (frameTransmissionCompoundObjID,
OPC_OBJTYPE_GENERIC, 0);

        op_ima_obj_attr_get (myObjectID, "Contention", &contentionCompoundObjID);
        contentionObjID = op_topo_child (contentionCompoundObjID, OPC_OBJTYPE_GENERIC, 0);


        receiverInterruptsSemafor = 0;
        receiverEndTime = 0.0;


        op_ima_obj_attr_get (myNodeObjectID, "name", myName);

        op_ima_obj_attr_get (myObjectID, "Range", &range);
        op_ima_obj_attr_get (myObjectID, "Error Rate", &errorRate);
        op_ima_sim_attr_get (OPC_IMA_DOUBLE, "Propagation Speed", &propagationSpeed);
        printf("\npropagation speed = %f", propagationSpeed);

        op_ima_obj_attr_get (myObjectID, "Outbound Channel Data Rate",
&outboundChannelDataRate);

        // Get the parameters grouped around the MAC parameters compound attribute
        op_ima_obj_attr_get (myObjectID, "MAC Address", &myAddress);
        op_ima_obj_attr_get (myObjectID, "Destination Node", &destinationAddress);

        // Get the parameters grouped around the frames size compound attribute
```

```
	op_ima_obj_attr_get (framesSizeObjID, "ACK Size", &sizeACK);
	op_ima_obj_attr_get (framesSizeObjID, "CTS Size", &sizeCTS);
	op_ima_obj_attr_get (framesSizeObjID, "RTS Size", &sizeRTS);
	op_ima_obj_attr_get (framesSizeObjID, "Data Frame Header Size",
&sizeDataFrameHeader);
	op_ima_sim_attr_get (OPC_IMA_INTEGER, "Data_Frame_Payload_Size",
&sizeDataFramePayload);

	// Get the parameters grouped around the contention compound attribute
	op_ima_obj_attr_get (contentionObjID, "Min Contention Window",
&minContentionWindow);
	op_ima_obj_attr_get (contentionObjID, "Max Contention Window",
&maxContentionWindow);

	op_ima_obj_attr_get (contentionObjID, "DIFS", &difsDuration);
	if (difsDuration == DOUBLE_MINUS_ONE)
		{
		difsDuration = (((double) sizeRTS) / outboundChannelDataRate) + (range /
propagationSpeed);
	printf("\nInside if DIFS = %f", difsDuration);
		}

	op_ima_obj_attr_get (contentionObjID, "SIFS", &sifsDuration);
	if (sifsDuration == DOUBLE_MINUS_ONE)
		sifsDuration = 0.01;

	op_ima_obj_attr_get (contentionObjID, "Slot Time", &slotDuration);
	if (slotDuration == DOUBLE_MINUS_ONE)
		slotDuration = (((double) sizeRTS) / outboundChannelDataRate) + (range /
propagationSpeed);

	printf("\noutboundchanneldatarate = %f", outboundChannelDataRate);
	printf("\nrange = %f", range);
	printf("\npropagation speed = %f", propagationSpeed);
	printf("\nDIFS = %f", difsDuration);
	printf("\nSIFS = %f", sifsDuration);
	printf("\nslot = %f", slotDuration);

	// Get the parameters grouped around the frame transmission compound attribute
	op_ima_obj_attr_get (frameTransmissionObjID, "Max Retransmission Attempts",
&maxRetransmissionAttempts);
	op_ima_obj_attr_get (frameTransmissionObjID, "Buffer Size",
&highLayerListMaxSize);
	op_ima_obj_attr_get (frameTransmissionObjID, "Retry Limit", &retryLimit);
	op_ima_obj_attr_get (frameTransmissionObjID, "Max Receive Lifetime",
&maxReceiveLifetime);


	// Obtain general attributes
	op_ima_sim_attr_get (OPC_IMA_INTEGER, "UAN_Network_Mode", &networkMode);
	op_ima_obj_attr_get (myObjectID, "UAN Type Of Node", &typeOfNode);

	// Register the log handles and related flags
	configLogHandle		= op_prg_log_handle_create (OpC_Log_Category_Configuration,
"Under Water Networks", "MAC "

"Configuration", 128);
	dropPacketLogHandle	= op_prg_log_handle_create (OpC_Log_Category_Protocol,
"Under Water Networks", "Data "
                                                                        "packet
Drop",  128);
	dropPacketEntryLogFlag = OPC_FALSE;

	// Creating list to store the information about the inbound channels
	channels_ptr = op_prg_list_create ();

	// Allocating memory for the flags used in this process model
	flags = (UwnT_Flags *) op_prg_mem_alloc (sizeof (UwnT_Flags));

	// Initially resetting all the flags
	flags->packet_to_send  = OPC_FALSE;
```

```
        flags->fragments_to_send = OPC_FALSE;
        flags->backoff_required              = OPC_FALSE;
        flags->perform_backoff = OPC_FALSE;
        flags->rts_sent                      = OPC_FALSE;
        flags->rcvd_bad_packet        = OPC_FALSE;
        flags->receiver_busy          = OPC_FALSE;
        flags->transmitter_busy           = OPC_FALSE;
        flags->immediate_xmt          = OPC_FALSE;
        flags->forced_bk_end      = OPC_FALSE;
        flags->nav_updated                   = OPC_FALSE;
        flags->collision                     = OPC_FALSE;

        // Initialize segmentation and reassembly buffers
        defragmentationListPtr = op_prg_list_create ();
        currentIndexDefragmentationBuffer = -1;
        commonRSMbufPtr        = op_sar_buf_create (OPC_SAR_BUF_TYPE_REASSEMBLY,
OPC_SAR_BUF_OPT_DEFAULT);

        // Registering local statistics
        packetLoadHandle                     = op_stat_reg ("UAN.Load (packets)",
                        OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);
        bitsLoadHandle                       = op_stat_reg ("UAN.Load (bits/sec)",
                                OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);
        highLayerPacketRcvd          = op_stat_reg ("UAN.Hld Queue Size (packets)",
          OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);
        backoffSlotsHandle                   = op_stat_reg ("UAN.Backoff Slots (slots)",
                  OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);
        dataTrafficSentHandle         = op_stat_reg ("UAN.Data Traffic Sent
(packets/sec)",          OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);
        dataTrafficRcvdHandle         = op_stat_reg ("UAN.Data Traffic Rcvd
(packets/sec)",          OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);
        dataTrafficSentHandleInBits   = op_stat_reg ("UAN.Data Traffic Sent (bits/sec)",
OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);
        dataTrafficRcvdHandleInBits   = op_stat_reg ("UAN.Data Traffic Rcvd (bits/sec)",
OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);
        ctrlTrafficSentHandle         = op_stat_reg ("UAN.Control Traffic Sent
(packets/sec)",  OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);
        ctrlTrafficRcvdHandle         = op_stat_reg ("UAN.Control Traffic Rcvd
(packets/sec)",  OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);
        ctrlTrafficSentHandleInBits   = op_stat_reg ("UAN.Control Traffic Sent
(bits/sec)",     OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);
        ctrlTrafficRcvdHandleInBits   = op_stat_reg ("UAN.Control Traffic Rcvd
(bits/sec)",     OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);
        dropPacketHandle                     = op_stat_reg ("UAN.Dropped Data Packets
(packets/sec)",  OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);
        dropPacketHandleInBits        = op_stat_reg ("UAN.Dropped Data Packets
(bits/sec)",     OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);
        retransHandle                        = op_stat_reg ("UAN.Retransmission Attempts
(packets)",   OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);
        mediaAccessDelay                     = op_stat_reg ("UAN.Media Access Delay
(sec)",                     OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);

        eteDelayHandle[1]                    = op_stat_reg ("UAN.Delay 1 (sec)",
                        OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);
        eteDelayHandle[2]                    = op_stat_reg ("UAN.Delay 2 (sec)",
                        OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);
        eteDelayHandle[3]                    = op_stat_reg ("UAN.Delay 3 (sec)",
                        OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);
        eteDelayHandle[4]                    = op_stat_reg ("UAN.Delay 4 (sec)",
                        OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);
        eteDelayHandle[5]                    = op_stat_reg ("UAN.Delay 5 (sec)",
                        OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);
        eteDelayHandle[6]                    = op_stat_reg ("UAN.Delay 6 (sec)",
                        OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);
        eteDelayHandle[7]                    = op_stat_reg ("UAN.Delay 7 (sec)",
                        OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);
        eteDelayHandle[8]                    = op_stat_reg ("UAN.Delay 8 (sec)",
                        OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);
        eteDelayHandle[9]                    = op_stat_reg ("UAN.Delay 9 (sec)",
                        OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);
        eteDelayHandle[10]                   = op_stat_reg ("UAN.Delay 10 (sec)",
```

```
                        OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);
        eteDelayHandle[11]                  = op_stat_reg ("UAN.Delay 11 (sec)",
                        OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);
        eteDelayHandle[12]                  = op_stat_reg ("UAN.Delay 12 (sec)",
                        OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);
        eteDelayHandle[13]                  = op_stat_reg ("UAN.Delay 13 (sec)",
                        OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);
        eteDelayHandle[14]                  = op_stat_reg ("UAN.Delay 14 (sec)",
                        OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);
        eteDelayHandle[15]                  = op_stat_reg ("UAN.Delay 15 (sec)",
                        OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);
        eteDelayHandle[16]                  = op_stat_reg ("UAN.Delay 16 (sec)",
                        OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);

        eteDelayBackgroundTraffic     = op_stat_reg ("UAN.Delay Background (sec)",
          OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);
        eteDelayNonPeriodicTraffic    = op_stat_reg ("UAN.Delay Non-Periodic (sec)",
                 OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);


        channelReservHandle                 = op_stat_reg ("UAN.Channel Reservation
(sec)",                 OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);
        throughputHandle                    = op_stat_reg ("UAN.Throughput (bits/sec)",
                 OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);

        // Registering global statistics
        globalETEdelayHandle          = op_stat_reg ("UAN Global.Delay (sec)",
              OPC_STAT_INDEX_NONE, OPC_STAT_GLOBAL);
        globalLoadHandle                    = op_stat_reg ("UAN Global.Load (bits/sec)",
                       OPC_STAT_INDEX_NONE, OPC_STAT_GLOBAL);
        globalThroughputHandle              = op_stat_reg ("UAN Global.Throughput
(bits/sec)",       OPC_STAT_INDEX_NONE, OPC_STAT_GLOBAL);
        globalDroppedDataHandle             = op_stat_reg ("UAN Global.Data Dropped
(bits/sec)",     OPC_STAT_INDEX_NONE, OPC_STAT_GLOBAL);
        globalMACdelayHandle          = op_stat_reg ("UAN Global.Media Access Delay
(sec)",    OPC_STAT_INDEX_NONE, OPC_STAT_GLOBAL);

        // Initialize retry and back-off slot counts
        retryCount   = 0;
        backoffSlots = BACKOFF_SLOTS_UNSET;

        /*
        Initialize the packet pointers that holds the last transmitted packets to be used
for retransmissions when
        necessary
        */
        transmitFrameCopyPtr = OPC_NIL;

        // Initialize received packets
        if (typeOfNode == UanE_Gateway_Node)
        {
                for (idx = 0; idx < MAX_NUMBER_NODES; idx++)
                {
                        allPacketsReceivedByNode[idx] = 0.0;
                        constPacketsReceivedByNode[idx] = 0;
                        eventTrafficReceivedByNode[idx] = 0;
                        eteDelayAllTrafficByNode[idx] = 0.0;
                        eteDelayConstTrafficByNode[idx] = 0.0;
                        eteDelayEventTrafficByNode[idx] = 0.0;
                        allTrafficReceivedByNode[idx] = 0.0;
                        constTrafficReceivedByNode[idx] = 0.0;
                        eventTrafficReceivedByNode[idx] = 0.0;

                        bitsGeneratedByNode[idx] = 0.0;
                        packetsGeneratedByNode[idx] = 0.0;
                        packetsInQueueByNode[idx] = 0;
                        packetsDroppedByNode[idx] = 0;
                        retransmissionAttemptsByNode[idx] = 0;
                        backoffSlotsByNode[idx] = 0.0;
                        lastTransmittedPacket[idx] = 0.0;
                        lastReceivedPacket[idx] = 0.0;
```

```
		}
	}

	strcpy(nodesName[myAddress], myName);



	// Initialize NAV time
	navTime = 0;

	/* Initialize receiver idle timer. */
	//	rcv_idle_time = -2.0 * difs_time;

	// Initializing the sum of sizes of the packets in the higher layer queue
	highLayerListTotalSize = 0;

	// Initialize the state variables related with the current frame that is being
handled
	currentPacketSize  = 0;
	receiveTime = 0.0;

	// Initializing frame response to send to none
	frameTypeToSend = UanE_None;

	// Initializing expected frame type to none
	expectedFrameType = UanE_None;

	// Set the variable that holds the current simulation time
	currentTime = op_sim_time ();

	/*
	Data arrived from higher layer is queued in the buffer. Pool memory is used for
allocating data structure for the
	higher layer packet. This structure is then inserted in the higher layer arrival
queue
	*/
	highLayerPMH = op_prg_pmo_define ("UAN high layer list elements", sizeof
(UanT_High_Layer_List_Elem), 32);
	highLayerListPtr = op_prg_list_create();

	/* Fragmentation buffer. The packet is fragmented and inserted into the buffer.
Pool memory is used
	   for allocating data structure for each fragment. This structure is then
inserted in the
	   fragmentation list */
	fragmentationPMH = op_prg_pmo_define ("UAN Fragments of a packet to send", sizeof
(UanT_Mac_Fragmentation_List_Element), 5);
	fragmentationList = op_prg_list_create();

	errorRateDistribution = op_dist_load ("uniform", 0.0, 1.0);


	FOUT;
}

static void updateControlTrafficStats (double rcvd_pk_size, double rx_start_time)
{

	FIN (updateControlTrafficStats (rcvd_pk_size, rx_start_time));

	/*Update received control traffic statistics. Write the appropriate values for
	  start and end of the reception */
	op_stat_write_t (ctrlTrafficRcvdHandleInBits, rcvd_pk_size, rx_start_time);
	op_stat_write   (ctrlTrafficRcvdHandleInBits, 0.0);
	op_stat_write_t (ctrlTrafficRcvdHandle,       1.0, rx_start_time);
	op_stat_write   (ctrlTrafficRcvdHandle,       0.0);

	FOUT;
}
```

```
static void updateDataTrafficStats (double rcvd_pk_size, double rx_start_time)
{

        FIN (updateDataTrafficStats (rcvd_pk_size, rx_start_time));

        /* Update received data traffic statistics. Write the appropriate values for start
           and end of the reception */
        op_stat_write_t (dataTrafficRcvdHandleInBits, rcvd_pk_size, rx_start_time);
        op_stat_write (dataTrafficRcvdHandleInBits, 0.0);
        op_stat_write_t (dataTrafficRcvdHandle, 1.0, rx_start_time);
        op_stat_write (dataTrafficRcvdHandle, 0.0);

        FOUT;
}

static void printStateVariables()
{
        FIN (printStateVariables());
        printf("\n\n#####################################################");
        printf("\nTIMES:");
        printf("\nretryCount    = %f", retryCount);
        printf("\nNAV           = %f", navTime);
        printf("\nCurrent Time  = %f", currentTime);
        printf("\nReceive Time  = %f", receiveTime);
        printf("\nrxEndTime     = %f", rxEndTime);
        printf( "\nReceiver Idle = %f", receiverIdleTime);
        printf("\n\nFRAME TYPES:");
        printf("\nFrame Type to send  = %s", frameType(frameTypeToSend));
        printf("\nExpected Frame Type = %s", frameType(expectedFrameType));
        printf("\nLast Frame TX Type  = %s", frameType(lastFrameTxType));
        printf("\n\nFLAGS:");
        printf("\nPacket to send      = %s", (flags->packet_to_send == OPC_TRUE) ? "TRUE"
: "FALSE");
        printf("\nFragments to send   = %s", (flags->fragments_to_send == OPC_TRUE) ?
"TRUE" : "FALSE");
        printf("\nBackoff required    = %s", (flags->backoff_required == OPC_TRUE) ?
"TRUE" : "FALSE");
        printf("\nRTS Sent            = %s", (flags->rts_sent == OPC_TRUE) ? "TRUE" :
"FALSE");
        printf("\nBad Packet received = %s", (flags->rcvd_bad_packet == OPC_TRUE) ?
"TRUE" : "FALSE");
        printf("\nReceiver Busy       = %s", (flags->receiver_busy == OPC_TRUE) ? "TRUE"
: "FALSE");
        printf("\n#####################################################\n\n");
        FOUT;
}


static char* frameType(int type)
{
        FIN (frameType(type));

        switch (type)
        {
                case UanE_Rts:
                        FRET ("RTS            ");

                case UanE_Cts:
                        FRET ("CTS            ");

                case UanE_Ack:
                        FRET ("ACK            ");

                case UanE_None_Transit:
                        FRET ("NONE TRANSIT   ");

                case UanE_Data:
                        FRET ("DATA           ");

                case UanE_None:
                        FRET ("NONE           ");
```

```
                case UanE_Rts_Ime:
                        FRET ("RTS IMEDIATELY  ");


                default:
                        FRET ("Wrong frame type");
        }
}


/*This function is called just before a frame received from physical layer being
forwarded to the higher layer to
update end-to-end delay and throughput statistics
*/
static void record_final_stats ()
{
        int idx;
        double globalAllTrafficDelay = 0.0;
        double globalConstTrafficDelay = 0.0;
        double globalEventTrafficDelay = 0.0;
        double globalGeneratedBits = 0.0;
        int globalGeneratedPackets = 0;
        int globalAllTrafficReceivedPackets = 0;
        int globalConstTrafficReceivedPackets = 0;
        int globalEventTrafficReceivedPackets = 0;
        double globalAllTrafficReceived =0.0;
        double globalConstTrafficReceived =0.0;
        double globalEventTrafficReceived =0.0;
        int globalPacketsDropped = 0;
        int globalPacketsInQueue = 0;
        int globalRetransmissionAttempts = 0;
        double globalBackoffSlotsPerformed = 0;
        int globalFramesCollided = 0;
        char stats[100];
        FIN (record_final_stats ());

        // Write the average bits per second generated by each node
        if (bitsGeneratedByNode[myAddress] != 0.0)
        {
                strcpy (stats, "");
                strcpy (stats, "Traffic Generated - Average (bits/sec) - ");
                strcat (stats, myName);
                op_stat_scalar_write (stats, bitsGeneratedByNode[myAddress] /
op_sim_time());
        }

        // Write the average packets per second generated by each node
        if (packetsGeneratedByNode[myAddress] != 0)
        {
                strcpy (stats, "");
                strcpy (stats, "Packets Generated - Average (packets/sec) - ");
                strcat (stats, myName);
                op_stat_scalar_write (stats, (double) packetsGeneratedByNode[myAddress] /
op_sim_time());
        }

        // Write the average retransmission attempts made by each node
        if (retransmissionAttemptsByNode[myAddress] != 0)
        {
                strcpy (stats, "");
                strcpy (stats, "Retransmission Attempts (retransmissions) - ");
                strcat (stats, myName);
                op_stat_scalar_write (stats, (double)
retransmissionAttemptsByNode[myAddress]);
        }

        // Write the average backoff slots performed by each node
        if (backoffSlotsByNode[myAddress] != 0.0)
        {
                strcpy (stats, "");
```

```
		strcpy (stats, "Backoff Slots Perfomed (slots) - ");
		strcat (stats, myName);
		op_stat_scalar_write (stats, backoffSlotsByNode[myAddress]);
	}

	// Write the average backoff slots performed by each node
	if (framesCollidedByNode[myAddress] != 0)
	{
		strcpy (stats, "");
		strcpy (stats, "Frames Collided (frames) - ");
		strcat (stats, myName);
		op_stat_scalar_write (stats, framesCollidedByNode[myAddress]);
	}

	// Write the number of packets dropped by each node
	if (typeOfNode != UanE_Gateway_Node)
	{
		strcpy (stats, "");
		strcpy (stats, "Packets Dropped (packets) - ");
		strcat (stats, myName);
		op_stat_write_scalar (stats, (double) packetsDroppedByNode[myAddress]);

		strcpy (stats, "");
		strcpy (stats, "Packets in Queue (packets) - ");
		strcat (stats, myName);
		op_stat_write_scalar (stats, (double) packetsInQueueByNode[myAddress]);
	}

	if (typeOfNode == UanE_Gateway_Node)
	{

		// Write descriminated stats and calculate global stats
		for (idx = 0; idx < MAX_NUMBER_NODES; idx++)
		{
			if (allPacketsReceivedByNode[idx] != 0)
			{

				// Write the end-to-end delay of all traffic descriminated
by generating node
				strcpy (stats, "");
				strcpy (stats, "End-to-End Delay - All Traffic - Average
(sec) - ");
				strcat (stats, nodesName[idx]);
				op_stat_scalar_write (stats, eteDelayAllTrafficByNode[idx]
/ (double) allPacketsReceivedByNode[idx]);


				if (constPacketsReceivedByNode[idx] != 0)
				{
					// Write the end-to-end delay of the background
traffic descriminated by generating node
					strcpy (stats, "");
					strcpy (stats, "End-to-End Delay - Background -
Average (sec) - ");
					strcat (stats, nodesName[idx]);
					op_stat_scalar_write (stats,
eteDelayConstTrafficByNode[idx] / (double) constPacketsReceivedByNode[idx]);
				}

				if (eventPacketsReceivedByNode[idx] != 0)
				{
					// Write the end-to-end delay of the non-periodic
traffic descriminated by generating node
					strcpy (stats, "");
					strcpy (stats, "End-to-End Delay - Non-Periodic -
Average (sec) - ");
					strcat (stats, nodesName[idx]);
					op_stat_scalar_write (stats,
eteDelayEventTrafficByNode[idx] / (double) eventPacketsReceivedByNode[idx]);
				}
```

```
                                // Write the total throughput descriminated by generating
node
                                strcpy (stats, "");
                                strcpy (stats, "Throughput - All Traffic - Average
(bits/sec) - From ");
                                strcat (stats, nodesName[idx]);
                                op_stat_scalar_write (stats, allTrafficReceivedByNode[idx]
/ op_sim_time());

                                // Write the background throughput descriminated by
generating node
                                strcpy (stats, "");
                                strcpy (stats, "Throughput - Background - Average
(bits/sec) - From ");
                                strcat (stats, nodesName[idx]);
                                op_stat_scalar_write (stats,
constTrafficReceivedByNode[idx] / op_sim_time());

                                // Write the non-periodic throughput descriminated by
generating node
                                strcpy (stats, "");
                                strcpy (stats, "Throughput - Non-Periodic - Average
(bits/sec) - From ");
                                strcat (stats, nodesName[idx]);
                                op_stat_scalar_write (stats,
eventTrafficReceivedByNode[idx] / op_sim_time());

                        }

                        globalAllTrafficDelay += eteDelayAllTrafficByNode[idx];
                        globalConstTrafficDelay += eteDelayConstTrafficByNode[idx];
                        globalEventTrafficDelay += eteDelayEventTrafficByNode[idx];

                        globalAllTrafficReceivedPackets += allPacketsReceivedByNode[idx];
                        globalConstTrafficReceivedPackets +=
constPacketsReceivedByNode[idx];
                        globalEventTrafficReceivedPackets +=
eventPacketsReceivedByNode[idx];

                        globalAllTrafficReceived += allTrafficReceivedByNode[idx];
                        globalConstTrafficReceived += constTrafficReceivedByNode[idx];
                        globalEventTrafficReceived += eventTrafficReceivedByNode[idx];

                        globalGeneratedBits += bitsGeneratedByNode[idx];
                        globalGeneratedPackets += packetsGeneratedByNode[idx];
                        globalPacketsDropped += packetsDroppedByNode[idx];
                        globalPacketsInQueue += packetsInQueueByNode[idx];
                        globalRetransmissionAttempts += retransmissionAttemptsByNode[idx];
                        globalBackoffSlotsPerformed += backoffSlotsByNode[idx];
                        globalFramesCollided += framesCollidedByNode[idx];
                }

                // Write global stats
                op_stat_scalar_write ("Traffic Generated - Average (bits/sec) - Global",
globalGeneratedBits / op_sim_time());
                op_stat_scalar_write ("Packets Generated - Average (packets/sec) -
Global", (double) globalGeneratedPackets / op_sim_time());

                if (globalAllTrafficReceivedPackets != 0)
                        op_stat_scalar_write ("End-to-End Delay - All Traffic - Average
(sec) - Global", globalAllTrafficDelay / (double) globalAllTrafficReceivedPackets);

                if (globalConstTrafficReceivedPackets != 0)
                        op_stat_scalar_write ("End-to-End Delay - Background - Average
(sec) - Global", globalConstTrafficDelay / (double) globalConstTrafficReceivedPackets);

                if (globalEventTrafficReceivedPackets != 0)
                        op_stat_scalar_write ("End-to-End Delay - Non-Periodic - Average
(sec) - Global", globalEventTrafficDelay / (double) globalEventTrafficReceivedPackets);
```

```
		op_stat_scalar_write ("Throughput - All Traffic - Average (bits/sec) -
Global", globalAllTrafficReceived / op_sim_time());
		op_stat_scalar_write ("Throughput - Background - Average (bits/sec) -
Global", globalConstTrafficReceived / op_sim_time());
		op_stat_scalar_write ("Throughput - Non-Periodic - Average (bits/sec) -
Global", globalEventTrafficReceived / op_sim_time());
		op_stat_scalar_write ("Packets Dropped (packets) - Global", (double)
globalPacketsDropped);
		op_stat_scalar_write ("Packets in Queue (packets) - Global", (double)
globalPacketsInQueue);
		op_stat_scalar_write ("Retransmission Attempts (retransmissions) -
Global", (double) globalRetransmissionAttempts);
		op_stat_scalar_write ("Backoff Slots Perfomed (slots) - Global",
globalBackoffSlotsPerformed);
		op_stat_scalar_write ("Frames Collided (frames) - Global",
globalFramesCollided);
		op_stat_scalar_write ("Propagation Speed", (double) propagationSpeed);

		// Write parameters stats
		op_stat_scalar_write ("Network Mode", networkMode);
		op_stat_scalar_write ("Data Frame Payload Size (bits)",
sizeDataFramePayload);


		// To print in debug mode
		for (idx = 1; idx < MAX_NUMBER_NODES; idx++)
		{
			printf("\n\nNode: %s\n", nodesName[idx]);
			printf("Pkts generated: %i\n", packetsGeneratedByNode[idx]);
			printf("Pkts in Queue: %i\n", packetsInQueueByNode[idx]);
			printf("Pkts Dropped: %i\n", packetsDroppedByNode[idx]);
			printf("Pkts Rcvd: %i\n", allPacketsReceivedByNode[idx]);
			printf("(Gen = Queue + drop + rcvd): %s\n",
packetsGeneratedByNode[idx] == packetsInQueueByNode[idx] + packetsDroppedByNode[idx] +
allPacketsReceivedByNode[idx] ? "TRUE" : "FALSE");
			printf("Last Sent Pkt: %f\n", lastTransmittedPacket[idx]);
			printf("Last Rcvd Pkt: %f\n", lastReceivedPacket[idx]);
		}

		printf("\n\nCompound Nodes 1, 2, 3, 4, 5, 6, 7:\n");

		printf("Pkts generated = %i\n", globalGeneratedPackets);

		printf("Pkts in transit = %i\n", globalPacketsInQueue);

		printf("Dropped pkts = %i\n", globalPacketsDropped);

		printf("Received Pkts = %i\n", globalAllTrafficReceivedPackets);


	}

	FOUT;
}


================================================================================
			Enter Execs for the unforced state "init"
================================================================================
/*
Initialization of the process model.

1. Initialize state variables
2. Read model attribute values in variables
3. Create global lists
4. Register statistics handlers
*/

modelInitialization();

op_intrpt_schedule_self(op_sim_time(), 0);
```

```
--------------------------------------------------------------------------------
================================================================================
                    Exit Execs for the unforced state "init"
================================================================================
NONE
--------------------------------------------------------------------------------
================================================================================
                          transition   init -> res_names
================================================================================
name:  tr_34
condition:
executive:
color:  RGB000
drawing style: spline
doc file:      pr_transition

--------------------------------------------------------------------------------


================================================================================
                    Enter Execs for the unforced state "idle"
================================================================================
/*
The purpose of this state is to wait until the packet has arrived from the
higher or lower layer. In this state
following intrpts can occur:
1. Data arrival from application layer
2. Frame (DATA,ACK,RTS,CTS) rcvd from PHY layer
3. Receiver On stating that frame is being rcvd
*/

// Determine if this is the end of simulation and record final stats
if (op_intrpt_type() == OPC_INTRPT_ENDSIM)
       record_final_stats();
--------------------------------------------------------------------------------
================================================================================
                    Exit Execs for the unforced state "idle"
================================================================================
// Interrupt processing routine
interrupts_process ();

/* Schedule deference interrupt when there is a frame to transmit at the stream interrupt
and the receiver is not busy */
if (NEED_TO_TRANSMIT)
{
       backoffSlots = BACKOFF_SLOTS_UNSET;
       schedule_deference ();
}
--------------------------------------------------------------------------------
================================================================================
                          transition   idle -> defer
================================================================================
name:  tr_15
condition:     NEED_TO_TRANSMIT
executive:
color:  RGB000
drawing style: line
doc file:      pr_transition

--------------------------------------------------------------------------------

================================================================================
                          transition   idle -> idle
================================================================================
name:  tr_31
condition:     default
executive:
color:  RGB000
drawing style: spline
doc file:      pr_transition
```

```
--------------------------------------------------------------------------------


================================================================================
                Enter Execs for the unforced state "transmit"
================================================================================
/*
In this state following intrpts can occur:
1. Data arrival from application layer
2. Frame (DATA,ACK,RTS,CTS) rcvd from PHY layer
3. Receiver ON stating that frame is being rcvd
4. Transmission completed intrpt from physical layer
Queue the packet for Data Arrival from the higher layer, and do not change state.
After Transmission is completed change state to FRM_END. No response is generated
for any lower layer packet arrival.
*/

if (flags->rcvd_bad_packet == OPC_FALSE && interruptType == OPC_INTRPT_SELF)
{
	if (interruptCode == UwnE_Deference_Off || interruptCode == UwnE_Backoff_Elapsed)
	{
		frame_transmit ();

	}
}

// Determine if this is the end of simulation and record final stats
if (op_intrpt_type() == OPC_INTRPT_ENDSIM)
	record_final_stats();
--------------------------------------------------------------------------------
================================================================================
                  Exit Execs for the unforced state "transmit"
================================================================================
// Check if the interrupt comes from the physical layer
if (op_intrpt_type() == OPC_INTRPT_REMOTE)
{
	/* If a packet is received while the station is transmitting then mark the packet
as bad. On the first
	   execution of the exit execs of this state, the flag receiver_busy is false. The
only interrupt that could
	   be received from the receiver is the Uwn_Receiver_On. This may change when we
call interrupts_process()
	   later on. Therefore the interrupt of interest - receiver on - is first
processed by the following code */
	if ( op_intrpt_code() == UwnE_Receiver_On)
		flags->rcvd_bad_packet = OPC_TRUE;

	/* On the other hand, if he interrupt received is from the transmitter, the only
one that we are interested
	   is in the end of transmission, because the beginning was already set in the
enter execs of this state */
	else if (op_intrpt_code() == UwnE_Transmitter_Off)
	{
		flags->transmitter_busy = OPC_FALSE;

		printf("\n#%s# - TRANSMITER OFF", myName);

		/* Also update the receiver idle time, since with the end of our
transmission, the medium may become idle
		   again */
		receiverIdleTime = op_sim_time ();
	}
}


// While transmitting, we received a packet from physical layer. Mark the packet as bad
else if ((op_intrpt_type () == OPC_INTRPT_STRM) && (op_intrpt_strm () ==
INPUT_STREAM_FROM_PHYSICAL_LAYER))
	flags->rcvd_bad_packet = OPC_TRUE;

/* Call the interrupt processing routine for each interrupt.*/
```

```
interrupts_process ();
--------------------------------------------------------------------------------
================================================================================
                        transition   transmit -> frm end
================================================================================
name:  tr_22
condition:     TRANSMISSION_COMPLETE
executive:
color: RGB000
drawing style: spline
doc file:      pr_transition

--------------------------------------------------------------------------------

================================================================================
                       transition   transmit -> transmit
================================================================================
name:  tr_32
condition:     default
executive:
color: RGB000
drawing style: spline
doc file:      pr_transition

--------------------------------------------------------------------------------


================================================================================
                    Enter Execs for the unforced state "defer"
================================================================================
/*
This state defer until the medium is available for transmission interrupts
that can occur in this state are:
1. Data arrival from application layer
2. Frame (DATA,ACK,RTS,CTS) rcvd from PHY layer
3. Receiver ON stating that frame is being rcvd
4. Collision intrpt stating that more than one frame is rcvd
5. Deference timer has expired (self intrpt)

For Data arrival from application layer queue the packet. Set Backoff flag if
the station needs to backoff after deference because the medium is busy.
If the frame is destined for this station then set frame to respond and
set a deference timer to SIFS. Set deference timer to SIFS and don't change
states. If receiver starts receiving more than one frame then flag the
received frame as invalid frame and set a deference to EIFS.
*/

// Determine if this is the end of simulation and record final stats
if (op_intrpt_type() == OPC_INTRPT_ENDSIM)
       record_final_stats();
--------------------------------------------------------------------------------
================================================================================
                    Exit Execs for the unforced state "defer"
================================================================================
/* Store the previous receiver status before processing the interrupt, which may
   change the status information */
pre_rx_status = flags->receiver_busy;

// Call the interrupt processing routine for each interrupt
interrupts_process ();

/* If the receiver is busy while the station is deferring then clear the self
   interrupt. As there will be a new self interrupt generated once the receiver
   becomes idle again */
if (flags->receiver_busy && (op_ev_valid (deferenceEVH) == OPC_TRUE))
       op_ev_cancel (deferenceEVH);

/* Update the value of the temporary bad packet flag, which is used in the
   FRAME_RCVD macro below */
bad_packet_rcvd = flags->rcvd_bad_packet;
```

```
// If the receiver became idle again schedule the end of the deference
if (flags->receiver_busy == OPC_FALSE && pre_rx_status == OPC_TRUE)
        schedule_deference ();

/* While we were deferring, if we receive a frame which requires a response,
then we need to re-schedule our end of deference interrupt. Similarly, we need
to re-schedule it if the received frame made us set our NAV to a higher value */
else if (FRAME_RCVD && (frameTypeToSend != UanE_None ||
             flags->nav_updated == OPC_TRUE) &&
             op_ev_valid (deferenceEVH) == OPC_TRUE)
{
        // Cancel the current event and schedule a new one
        op_ev_cancel (deferenceEVH);
        schedule_deference ();
}
--------------------------------------------------------------------------------
================================================================================
                          transition   defer -> bkoff eva
================================================================================
name:  tr_17
condition:     DEFERENCE_OFF
executive:
color: RGB000
drawing style: spline
doc file:      pr_transition

--------------------------------------------------------------------------------

================================================================================
                            transition   defer -> defer
================================================================================
name:  tr_36
condition:     default
executive:
color: RGB000
drawing style: spline
doc file:      pr_transition

--------------------------------------------------------------------------------


================================================================================
                   Enter Execs for the forced state "bkoff eva"
================================================================================
/*
In this state we determine whether a back-off is necessary for the frame we are
trying to transmit. It is needed when station preparing to transmit frame
discovers that the medium is busy or when the station infers collision. Backoff
is not needed when the station is responding to the frame. Following a
successful packet transmission, again a back-off procedure is performed.
If backoff needed then check whether the station completed its backoff in the
last attempt. If not then resume the backoff from the same point, otherwise
generate a new random number for the number of backoff slots
*/

// Checking whether backoff is needed or not.
if (flags->perform_backoff == OPC_TRUE)
{
        if (backoffSlots == BACKOFF_SLOTS_UNSET)
        {
                /* Compute backoff interval using binary exponential process. After a
                   successful transmission we always use cw_min */
                if (retryCount <= 1)
                {
                        /* If retry count is set to 0 then set the maximum backoff slots
                        to min window size */
                        maxBackoff = minContentionWindow + 1;
                }

                if (retryCount > 0)
                {
```

```
                        // We are retransmitting. Increase the back-off window size
                        maxBackoff = maxBackoff * 2 + 1;
                }

                // The number of possible slots grows exponentially until it exceeds a
fixed limit
                if (maxBackoff > maxContentionWindow)
                        maxBackoff = maxContentionWindow;

                /* Obtain a uniformly distributed random integer between 0 and the minimum
contention window size. Scale
                   the number of slots according to the number of retransmissions */
                backoffSlots = floor (op_dist_uniform ((double) maxBackoff));

                // Reporting number of backoff slots as a statistic
                op_stat_write (backoffSlotsHandle, backoffSlots);
                backoffSlotsByNode[myAddress] += backoffSlots;

        }

        // Set a timer for the end of the backoff interval
        interruptTime = (currentTime + backoffSlots * slotDuration);

        if (networkMode == UanE_Aloha_Alike)
        {
                if (interruptTime < currentTime + sifsDuration)
                        interruptTime = currentTime + sifsDuration;
        else
                if (interruptTime < currentTime + difsDuration)
                        interruptTime = currentTime + difsDuration;
        }

        backoffElapsedEVH = op_intrpt_schedule_self (interruptTime, UwnE_Backoff_Elapsed);

}

// Determine if this is the end of simulation and record final stats
if (op_intrpt_type() == OPC_INTRPT_ENDSIM)
        record_final_stats();
--------------------------------------------------------------------------------
================================================================================
                  Exit Execs for the forced state "bkoff eva"
================================================================================
printf("\n\n LEAVING Backoff evaluation State");
printStateVariables();
--------------------------------------------------------------------------------
================================================================================
                        transition   bkoff eva -> backoff
================================================================================
name:  tr_16
condition:     PERFORM_BACKOFF
executive:
color:  RGB000
drawing style: spline
doc file:      pr_transition

--------------------------------------------------------------------------------

================================================================================
                        transition   bkoff eva -> transmit
================================================================================
name:  tr_18
condition:     TRANSMIT_FRAME
executive:
color:  RGB000
drawing style: spline
doc file:      pr_transition

--------------------------------------------------------------------------------
```

```
================================================================================
                  Enter Execs for the unforced state "backoff"
================================================================================
/*
Processing Random Backoff
In this state following intrpts can occur:
1. Data arrival from application layer
2. Frame (DATA,ACK,RTS,CTS) rcvd from PHY layer
3. Receiver ON stating that frame is being rcvd

*/

// Determine if this is the end of simulation and record final stats
if (op_intrpt_type() == OPC_INTRPT_ENDSIM)
       record_final_stats();
--------------------------------------------------------------------------------
================================================================================
                  Exit Execs for the unforced state "backoff"
================================================================================
pre_rx_status = flags->receiver_busy;

// Call the interrupt processing routine for each interrupt
interrupts_process ();

// Set the number of slots to zero, once the backoff is completed
if (BACKOFF_COMPLETED)
{
       flags->backoff_required = OPC_FALSE;
       flags->perform_backoff = OPC_FALSE;
       backoffSlots = BACKOFF_SLOTS_UNSET;
}

// Pause the backoff procedure if our receiver just became busy
if (flags->receiver_busy == OPC_TRUE && pre_rx_status == OPC_FALSE)
{
       // Computing remaining backoff slots for next iteration
       backoffSlots =  ceil ((interruptTime - currentTime - PRECISION_RECOVERY) /
                                  slotDuration);

       /* Don't cancel the end-of-backoff interrupt if we have already    completed
          all the slots of the back-off */
       if (op_ev_valid (backoffElapsedEVH) == OPC_TRUE && backoffSlots > 0.0)
       {
              // Clear the self interrupt as station needs to defer
              op_ev_cancel (backoffElapsedEVH);

       }
}
--------------------------------------------------------------------------------
================================================================================
                          transition   backoff -> idle
================================================================================
name:  tr_19
condition:     BACK_TO_IDLE
executive:
color:  RGB000
drawing style:  spline
doc file:      pr_transition

--------------------------------------------------------------------------------

================================================================================
                         transition   backoff -> defer
================================================================================
name:  tr_21
condition:     BACK_TO_DEFER
executive:     schedule_deference()
color:  RGB000
drawing style:  line
doc file:      pr_transition
```

```
--------------------------------------------------------------------------------

================================================================================
                        transition   backoff -> backoff
================================================================================
name:  tr_27
condition:     default
executive:
color:  RGB000
drawing style: spline
doc file:      pr_transition

--------------------------------------------------------------------------------


================================================================================
                    Enter Execs for the forced state "frm end"
================================================================================
/*
The purpose of this state is to determine the next unforcedstate after completing
transmission. 3 cases:
1. If just transmitted RTS or DATA frame then wait for response with expected_frame_type
variable set and change
   the states to Wait for Response otherwise just DEFER for next transmission
2. If expected frame is rcvd then check to see what is the next frame to transmit and set
appropriate deference
   timer:
  2a. If all the data fragments are transmitted then check whether the queue is empty or
not. If not then based
      on threshold fragment the packet and based on threshold decide whether to send RTS
or not. If there is a
      data to be transmitted then wait for DIFS duration before contending for the
channel. If nothing to
      transmit then go to IDLE state and wait for the packet arrival from higher or lower
layer.
3. If expected frame is not rcvd then infer collision, set backoff flag, if retry limit
is not reached
   retransmit the frame by contending for the channel. If there is no frame expected then
check to see if there
   is any other frame to transmit.
*/

if (expectedFrameType == UanE_None || expectedFrameType == UanE_None_Transit)
{
        /* If the frame needs to be retransmitted or there is something in the
fragmentation buffer to transmit or the
           station needs to respond to a frame then schedule deference */
        if (FRAME_TO_TRANSMIT)
        {

                /* Schedule deference before frame transmission.            */
                schedule_deference ();
        }
}

/* The station needs to wait for the expected frame type So it will set the frame timeout
interrupt which will be
   executed if no frame is received in the set duration. */

else
{

        if (navTime < currentTime)
                navTime = currentTime;

        // Expecting a Cts after sending a Rts or expecting an Ack after sending all the
data frames
        if (expectedFrameType == UanE_Cts || expectedFrameType == UanE_Ack)
        {

                /* Schedule the timeout interrupt to the final of the navTime plus s
```

```
sifsDuration */
                timeout = 2 * ((((double) sizeRTS) / outboundChannelDataRate) + (range /
propagationSpeed)) + sifsDuration + slotDuration;
                frameTimeoutEVH = op_intrpt_schedule_self (op_sim_time() + timeout,
UwnE_Frame_Timeout);
        }

        // Expecting data after sending a Cts
        else if (expectedFrameType == UanE_Data)
        {
                timeout = (((double) sizeACK) / outboundChannelDataRate) + (range /
propagationSpeed);
                frameTimeoutEVH = op_intrpt_schedule_self (navTime - timeout ,
UwnE_Frame_Timeout);
        }
}

// Determine if this is the end of simulation and record final stats
if (op_intrpt_type() == OPC_INTRPT_ENDSIM)
        record_final_stats();
--------------------------------------------------------------------------------
================================================================================
                    Exit Execs for the forced state "frm end"
================================================================================
printf("\n\n LEAVING FRM END state");
printStateVariables();
--------------------------------------------------------------------------------
================================================================================
                          transition   frm end -> defer
================================================================================
name:  tr_25
condition:     FRM_END_TO_DEFER
executive:
color:  RGB000
drawing style:  spline
doc file:      pr_transition

--------------------------------------------------------------------------------

================================================================================
                           transition   frm end -> idle
================================================================================
name:  tr_26
condition:     FRM_END_TO_IDLE
executive:
color:  RGB000
drawing style:  spline
doc file:      pr_transition

--------------------------------------------------------------------------------

================================================================================
                         transition   frm end -> wait_frm
================================================================================
name:  tr_44
condition:     EXPECTING_FRAME
executive:
color:  RGB000
drawing style:  spline
doc file:      pr_transition

--------------------------------------------------------------------------------


================================================================================
                  Enter Execs for the unforced state "res_names"
================================================================================
op_intrpt_schedule_self(op_sim_time(),0);
--------------------------------------------------------------------------------
================================================================================
                  Exit Execs for the unforced state "res_names"
```

```
================================================================================
if (typeOfNode != UanE_Gateway_Node)
	findDestinationNodeName();
--------------------------------------------------------------------------------
================================================================================
                        transition   res_names -> idle
================================================================================
name:  tr_35
condition:
executive:
color:  RGB000
drawing style: spline
doc file:      pr_transition

--------------------------------------------------------------------------------


================================================================================
                Enter Execs for the unforced state "wait_frm"
================================================================================
/** The purpose of this state is to wait for the response after     **/
/** transmission. The only frames which require                                **/
/** acknowledgements are RTS and DATA frame.                                   **/
/** In this state following intrpts can occur:                                 **/
/** 1. Data arrival from application layer                                     **/
/** 2. Frame (DATA,ACK,RTS,CTS) rcvd from PHY layer                      **/
/** 3. Frame timeout if expected frame is not rcvd                       **/
/** 4. Busy intrpt stating that frame is being rcvd                **/
/** 5. Collision intrpt stating that more than one frame is rcvd**/
/** Queue the packet as Data Arrives from application layer         **/
/** If Rcvd unexpected frame then collision is inferred and         **/
/** retry count is incremented
	**/
/** if a collision stat interrupt from the rcvr then flag the   **/
/** received frame as bad
	**/


// Determine if this is the end of simulation and record final stats
if (op_intrpt_type() == OPC_INTRPT_ENDSIM)
	record_final_stats();
--------------------------------------------------------------------------------
================================================================================
                 Exit Execs for the unforced state "wait_frm"
================================================================================
/* Clear the frame timeout interrupt once the receiver is busy or the frame is received
(in case of collisions, the frames whose reception has started while we were transmitting
are excluded in the FRAME_RCVD macro) */
if (op_intrpt_type() == OPC_INTRPT_SELF && op_intrpt_code() == UanE_Resume_Timeout &&
op_ev_valid (frameTimeoutEVH))
{
	printf("\n#%s# - Timeout reset", myName);
	op_ev_cancel (frameTimeoutEVH);
}

// Call the interrupt processing routine for each interrupt request
interrupts_process ();

/* If expected frame is not received in the set duration or there is a collision at the
receiver then set the expected frame type to be none because the station needs to
retransmit the frame */
if (interruptType == OPC_INTRPT_SELF && interruptCode == UwnE_Frame_Timeout)
{
		printf("\n#%s# - Timeout occurred", myName);


	// If we are the sending node
	if (expectedFrameType == UanE_Cts || expectedFrameType == UanE_Ack)
	{
		// If we timeout we need to perform backoff
		flags->backoff_required = OPC_TRUE;
```

```
                /* Increment the retransmission counter and check whether further
                   retries are possible or the packet, or its remains, needs to be
discarded */
                retryCount++;
                frame_discard ();

                /* Reset the rts_sent flag in case we didn't receive an     ACK for our
data transmission
                   in spite of a successful RTS/CTS frame exchange */
                flags->rts_sent = OPC_FALSE;

                // Setting expected frame type to none frame
                expectedFrameType = UanE_None;

                frameTypeToSend = UanE_None;


                // Reset the NAV duration so that the retransmission is not unnecessarily
delayed
                navTime = currentTime;
                flags->nav_updated = OPC_TRUE;

                /* Determine if there are fragments to transmit. We need to transmit
                        them again */

                for (idx = 0; idx < op_prg_list_size (fragmentationList); idx++)
                {
                        sentFragment = (UanT_Mac_Fragmentation_List_Element*)
                                op_prg_list_access (fragmentationList, idx);

                        if (networkMode == UanE_Contention_Based)
                                sentFragment->transmitted = OPC_FALSE;

                        }
        }

        // If we are the receiving node
        else if (networkMode == UanE_Contention_Based && expectedFrameType == UanE_Data)
        {
                if (currentIndexDefragmentationBuffer != -1)
                {
                        frameTypeToSend = UanE_Ack;
                        expectedFrameType = UanE_None_Transit;
                }
                else
                {
                        frameTypeToSend = UanE_None;
                        expectedFrameType = UanE_None;

                        if (flags->packet_to_send == OPC_TRUE || flags->fragments_to_send
== OPC_TRUE)
                                frameTypeToSend = UanE_Rts;

                        // Reset the NAV duration so that the retransmission is not
unnecessarily delayed
                        navTime = currentTime;
                        flags->nav_updated = OPC_TRUE;
                }
        }

}
--------------------------------------------------------------------------------
================================================================================
                        transition   wait_frm -> frm end
================================================================================
name:  tr_43
condition:     FRAME_TIMEOUT || RESUME_TIMEOUT
executive:
color: RGB000
drawing style: spline
```

```
doc file:      pr_transition

-------------------------------------------------------------------------------

===============================================================================
                       transition   wait_frm -> wait_frm
===============================================================================
name:  tr_52
condition:     default
executive:
color: RGB000
drawing style: spline
doc file:      pr_transition

-------------------------------------------------------------------------------
```

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

[Gibson 2002] John Gibson, A. Larraza, K. Smith, Geoffrey Xie, "On the Impacts and Benefits of Implementing Full-Duplex Communications Links in an Underwater Acoustic Network", Proceedings of the 5th International Symposium on Technology and the Mine Problem, April 2002

[Gibson 2005a] John Gibson, Geoffrey Xie, Andy Kaminski. "Demand Assigned Channel Allocation Applied to Full-Duplex Underwater Acoustic Networking", unpublished work - pending publication by PACON International (2005)
-- PaACON: Pacific Congress (consortium of industry and academia to advance the study and economic development of Asia-Pacific maritime resources

[Gibson 2005b] John Gibson, Geoffrey Xie, José Coelho, Leopoldo Diaz-Gonzalez. "The impact of Contention Resolution verses a priori Channel Allocation on Latency in a Delay Constrained Network", Pending publication by the Undersea Networking Working Group, in support of The Technical Cooperative Program (TTCP)

[Hartfield 2003] Grant I. Hartfield. "Link-Layer and Network-Layer Performance of an Undersea Acoustic Network at Fleet Battle Experiment - India", Master's Thesis (MS-ISO), Naval Postgraduate School, June 2003

[IEEE 802.3 2002] Ethernet Working Group. "Part 3: Carrier Sense Multiple Access with Collision Detection (CSMA/CD) access method and physical layer specifications", IEEE Std 802.3-2002, Institute of Electric and Electronics Engineers (IEEE), March 2002 http://standards.ieee.org/getieee802/download/802.3-2002.pdf accessed on Mar 06, 2005.

[IEEE 802.11 1999] Wireless Local Area Network Working Group. "Part 11: Wireless Local Area Network Medium Access Control (MAC) and Physical Layer (PHY) specifications", 1999 Edition - Reaffirmed Jun 12, 2003, Institute of Electric and Electronics Engineers (IEEE), 1999.
http://standards.ieee.org/getieee802/download/802.11-1999.pdf accessed on Mar 06, 2005.

[Kurose 2003] James F. Kurose and Keith W. Ross, "Computer Networking – A Top-Down Approach Featuring the Internet", 2nd Edition, Addison Wesley, 2003

[Rice 2002] Joseph Rice, "Undersea Networked Acoustic Communication and Navigation for Autonomous Mine-Countermeasure Systems", Proceedings of the 5th International Symposium on Technology and the Mine Problem, April 2002

[Tanenbaum 2003] Andrew S. Tanenbaum. "Computer Networks", 4th Edition, Pearson Education Inc, 2003

[Xie 2001] Geoffrey Xie, John Gibson. "A Network Layer Protocol for UANs to Address Propagation Delay Induced Performance Limitations", Proceedings of the MTS/IEEE Oceans Conference, November 2001

[Xie 2004] Geoffrey Xie, John Gibson, and Kurtulus Bektas. "Evaluating the Feasibility of Establishing Full-Duplex Underwater Acoustic Channels", Proceedings of the Third Annual Mediterranean Ad Hoc Networking Workshop, June 2004

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Professor Su Wen
   Naval Postgraduate School
   Monterey, California

4. Professor Geoffrey Xie
   Naval Postgraduate School
   Monterey, California

5. Professor John Gibson
   Naval Postgraduate School
   Monterey, California